RL-TR-91-36, Vol IIb (of seven)
Final Technical Report
April 1991

ROMULUS: A COMPUTER SECURITY PROPERTIES MODELING ENVIRONMENT Mathesis

ORA

Ian Sutherland, Tanya Korelsky, Daryl McCullough,
David Rosenthal, Jonathan Seldin, Marcos Lam,
Carl Eichenlaub, Bruce Esrig, James Hook, Carl Klapper,
Garrel Pottinger, Owen Rambow, Stanley Perlo

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-91-36, Volume IIb (of seven) has been reviewed and is approved for publication.

APPROVED: Joseph W. Frank

JOSEPH W. FRANK
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Ronald Raposo

RONALD RAPOSO
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (COAC) Griffiss AFB, NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1991	Final Apr 85 - May 90

4. TITLE AND SUBTITLE
ROMULUS: A COMPUTER SECURITY PROPERTIES MODELING ENVIRONMENT, Mathesis

5. FUNDING NUMBERS
C - F30602-85-C-0098
PE - 35167G
PR - 1065
TA - 01
WU - 02

6. AUTHOR(S) Ian Sutherland, Tanya Korelsky, Daryl McCullough, David Rosenthal, Jonathan Seldin, Marcos Lam, Carl Eichenlaub, Bruce Esrig, James Hook, Carl Klapper, Garrel Pottinger, Owen Rambow, Stanley Perlo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ORA
301A Harris B. Dates Drive
Ithaca NY 14850-1313

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (COAC)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-91-36, Vol IIb (of seven)

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Joseph W. Frank/COAC/(315) 330-2925

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Romulus Report describes the Romulus Computer Security Properties Modeling Environment. Romulus is an environment and methodology for the modeling, analysis, and verification of trusted computer systems, together with supporting tools. The Romulus methodology is based on a mathematical theory of security developed at Odyssey Research Associates. The theory formalizes multilevel information flow security by introducing restrictiveness, a hookup security property. This means that a collection of secure restrictive composite system. Because of its composability, restrictiveness is a useful security property for large, complex, distributed systems.

Volume I presents an overview of the important ideas and tools incorporated into the Romulus system. Volume II describes the underlying theory of security as well as Mathesis, the mathematical foundations of Romulus.

NOTE: Rome Laboratory/RL (formerly Rome Air Development Center/RADC)

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computer Security, Romulus, Verification, Multilevel Security, Hookup Security	184
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Acknowledgments

I would like to thank Richard Platek, Garrel Pottinger, Tatiana Korelsky, and James Hook for their many helpful comments and suggestions. Garrel Pottinger was especially helpful in checking carefully the proof of the strong normalization theorem in Chapter 4. Richard Platek wrote part of the Introduction.

Very special thanks are due to Owen Rambow for his creative work in translating this work from its original form (written in 1st Word on an Atari ST) into LaTeX, and to Donna Simmons and Carlos Maymi for helping him.

Jonathan P. Seldin

Ithaca, New York
April 24, 1987

Contents

INTRODUCTION

This work is an introduction to MATHESIS, the underlying mathematical foundation for ROMULUS. In ROMULUS one proves that models, designs and formal specifications of information processing systems have security properties. For this to be meaningful it is essential that the underlying automated mathematical foundation itself be sound. It is a known fact that various design and program verification environments in widespread use within the computer security community have faulty logics and implementations; a knowledgeable user of these environments can exploit these flaws to prove false facts about system. A less malicious user could inadvertently exploit these flaws and also prove false facts about systems. Machine certification of proofs is thus called into question when the certification mechanisms themselves are not appropriately certified.

There are two basic explanations of these flaws. First, the informal theory which stands logically prior to the theorem prover has not been adequately worked out. The purpose of this document is to work such a theory for the ROMULUS mathematical component. In particular, we prove the formal consistency to this theory.

A second source of error occurs during implementation. Many automated mathematical components and theorem provers evolve incrementally; new features are continually added to make the theorem prover ever more powerful. Also specific algorithms are replaced by more efficient ones. This maintenance, like most software maintenance, is usually done in an *ad hoc* manner. Logical flaws have a way of slipping in during such improvements. Our approach to this problem is to provide a mathematical foundation which in principal is much stronger than presently needed. The underlying logic is a true mathematical foundation in that the usual mathematical entities, viz. sets, sequences, functions, relations, etc., are all definable in terms of our ground entities. Future extensions of the theorem prover consist in adding definitions to the basic logic. The standard basic theorems about the new entities (what are usually called axioms) are then provable in the basic logic.

We thus have two requirements for a mathematical foundation for verification:

the informal theory needs to be worked out prior to implementation; the foundational theory should be strong enough to support definitional extensions which will encompass a significant amount of mathematics. Several approaches to foundations satisfy these requirements. Our specific choice was determined by several further requirements. First, in order to add confidence to the correctness of the implementation it would be desirable that the underlying foundations have as few moving parts as possible; i.e. the number of basic entities, constructors, axioms, etc. be small. Second, it would be desirable for the foundation to have computational content. That is, within the logic mechanically decidable statements should be distinguishable from undecidable ones and when statements are decidable the decision procedures encoded in their proofs should be available as computer programs. Logicians with a strictly mathematical background have not required this distinction; in computer science it separates the possible from the impossible. The natural logic for such computable entities is called constructive logic. There are cases where classical logic differs from constructive logic; namely some classically valid proofs cannot be made in constructive logic. On the other hand, there is an important sense in which constructive logic is stronger than classical logic since the latter can be interpreted in the former.

Since constructive logic is not well-known outside of certain subfields of mathematics and computer science, a few words about it may be in order. If one proves in constructive logic that something exists, then one must either give an explicit construction of that thing or else give a set of directions for constructing it. It follows from this that although in classical logic one is concerned only with truth and not how that truth is established, in constructive logic one is concerned with provability and one takes nothing to be true unless one actually has or can obtain access to a proof of it. This requires the denial of the law of excluded middle: A or not A. For if A is a statement that something exists, then A or not A means that either there is a set of directions for constructing that thing, or else there is a proof that there can be no such set of directions; this is clearly not true. This makes constructive logic seem a bit strange to those who are not used to it. Since constructive logic was first used in mathematics as one reaction to the paradoxes of set theory and logic which were discovered at the turn of the century, most examples of the difference between constructive and classical logic have generally been mathematical examples. Such examples can be found, among other places, at the beginning of [Bee85], which also has other references.

It might be worthwhile here to look at a nonmathematical example. The law of excluded middle might well lead a legislator to propose a criminal law in which there is one penalty for a crime if A is true of the particular case and a different penalty if A is false. In classical logic, one is justified in concluding that if the crime covered

by the law is committed and there is a conviction, then one of the two penalties would be applied. But in practice this does not follow. For suppose it turns out to be extremely difficult for the court system to decide whether or not A is true in a particular case. Then the case may be appealed all the way to the Supreme Court, a process which can take years (even more than a decade). During this time, neither penalty will be applied. And the courts may wind up deciding that A is so difficult to decide that the courts cannot do so constitutionally (as they might, for example, if they conclude as a matter of fact that trying to decide A is so difficult that it is impossible to do so in a way that does not treat people arbitrarily); in this case, the original law would be unconstitutional, and so no penalty would be applied (even if it were not in dispute that the defendant had committed the crime). Here is a nonmathematical case in which the law of excluded middle can be doubted.

Note the relationship between the use of constructive logic and the need to consider how a decision can be made. Constructive logic is often thought of as the logic of what can actually be done by computations if there are no limitations of time and space, and this makes it particularly appropriate for reasoning about computing in a general setting. In fact, this connection is the basis of Constable's Nuprl proof development system, in which executable programs are generated by proving mathematical theorems[C*86].

Because we are interested in a proof system, we are especially interested in referring to proofs. A good system of constructive logic in which proofs are mentioned explicitly is the theory of constructions of Coquand [Coq85][1]. This is a system of type assignment to λ-terms; the proofs are (roughly) represented by the terms and the formulas by the types. Although the rules of the system are easy to state, the system is, in fact, the result of a considerable evolution through a number of other systems of typed λ-calculus, and is best understood in the light of those systems.

For this reason we shall not take up the theory of constructions itself until Chapter 4. In Chapter 1 we shall take a look at typed λ-calculus. In Chapter 2 we shall consider deductive systems which assign types to λ-terms without types. We shall consider the basic system and and several of its generalizations. These generalizations include the second-order polymorphic typed λ-calculus[2], Martin-Löf's theory of types[3], and generalized type assignment in the style of [HS86] Chapter 16. The theory of constructions is a form of generalized type assignment, and so readers will be in a position at the end of Chapter 2 to proceed directly to the theory itself in Chapter 4.

[1] See also [CH84], [CH86], [CH], [Coq86a], [Coq86b], and [Coq].

[2] This system was introduced independently by Girard [Gir71] and Reynolds [Rey74] and studied extensively by a number of people, including [FLO83].

[3] See [Mar75], [Mar82], [Mar84], Chapter XI of [Bee85], and [C*86].

However, to fully appreciate the theory of constructions, it is desirable to consider both constructive logic and the idea of interpreting terms as proofs and types as formulas. This idea, which is often called the *Curry-Howard isomorphism*, was introduced by a number of people independently, including [How80], who based the idea on an observation of Curry [CF58], §9E. We take up this subject in Chapter 3. We begin in Sections 3.1-3.2 with a simple calculus of constructive logic for implication formulas, and show its relation to the simple system of type assignment. We then proceed in Sections 3.3-3.4 to extend the system to the other propositional connectives, and show that the law of excluded middle fails in this calculus of constructive logic. This is enough of the chapter for a basic understanding of both constructive logic and the Curry-Howard isomorphism, and many readers may want to proceed directly from the end of section 3.4 to Chapter 4. However, some readers may want to see a treatment of predicate logic, and in Sections 3.4 and 3.5, we present versions of (constructive) first-order predicate logic and higher-order predicate logic which illustrate the Curry-Howard isomorphism and look toward one of Coquand's motivations for creating the theory of constructions.

In Chapter 4, we come to the theory of constructions itself. We give its rules in a natural deduction formulation, which is a bit different from the form in which Coquand gave them but is more closely associated with the systems of type assignment mentioned in Chapter 2. We then proceed to prove the main consistency theorem for the system, the *strong normalization theorem*. We next show the relationship between the natural deduction formulation given here and the original formulation of Coquand.

Finally, in Chapter 5, we take up the representation of logic and mathematics in the theory of constructions. This is clearly necessary if this theory is to serve as the mathematical basis for MATHESIS and the rest of the ROMULUS project. We show how to represent logic, both constructive and classical, natural numbers, sets, and functions. The representation of natural numbers includes a representation of the principle of mathematical induction, and the method of doing this can easily be extended to other inductively defined free algebras. As an example of this, we show how to represent lists (finite sequences); this representation has direct application to the formulation of the hook-up security theory which is used in ROMULUS. The material of this chapter is all based on the work of Coquand and Huet[4], but in addition to the definitions and examples of the papers of Coquand and Huet, we feel a need to use the strong normalization theorem to give some proofs that the representations of logical and mathematical concepts really behave correctly.

[4]See [CH86] and [CH] in particular.

Chapter 1

TYPED LAMBDA-CALCULUS

The λ-calculus is a fundamental prototype for functional programming languages, and the typed λ-calculus is the natural typed version. Here we shall consider as much of the typed λ-calculus as we will need for the rest of the work. A general introduction to both the λ-calculus and the typed λ-calculus can be found in Hindley & Seldin [HS86].

Most of the systems we will consider will not have models in the usual set-theoretic sense of that term. However, ordinary typed λ-calculus does have such models, and so we shall begin with them.

1.1 Type symbols and type structures.

Types are used for various kinds of data structures in different programming languages. Here, we will be concerned with certain particular compound type structures which are fairly common. They are: 1) the *function space type* $\alpha \to \beta$ of functions with arguments in α and values in β, 2) the *cartesian product* $\alpha \times \beta$ of two types α and β, and 3) the *disjoint sum* $\alpha + \beta$ of two types α and β.

For some purposes, the only kind of compound type we will be interested in will be the function space type. In other cases we will be interested in all three kinds of compound types. This leads to the two kinds of type symbols in the following definition:

Definition 1.1 (Ty$_\rho$ symbol) Assume that we have (finitely or countably many) atomic type symbols $\theta_1, \ldots, \theta_n, \ldots$. Then *basic type symbols* are defined as follows:

(a) Every atomic type symbol is a type symbol; and

(b) If α and β are type symbols, then so is $(\alpha \to \beta)$.

Extended type symbols are defined by (a) and:

(c) If α and β are type symbols, then so are $(\alpha \to \beta)$, $(\alpha \times \beta)$ and $(\alpha + \beta)$.

Remark It might appear that the basic type symbols limit us to functions of one variable. This appearance is false, for functions of several variables can by reduced to functions of one variable by a process known as *currying* (after H. B. Curry, who used it extensively; actually the process was used by others before Curry). To see how currying works, consider the example

$$h(x, y) = x - y.$$

Let h^* be the one-place function whose value $h^*(a)$ at an argument a is defined to be the function

$$f(y) = a - y = h(a, y).$$

Then we have

$$h^*(a)(y) = h(x, y),$$

and we have replaced our original two-place function by a new function of one variable. Our notation will reflect the process of currying, since

$$\alpha_1 \to \alpha_2 \to \ldots \alpha_{n-1} \to \alpha_n$$

will be an abbreveation for

$$\alpha_1 \to (\alpha_2 \to (\ldots(\alpha_{n-1} \to \alpha_n)\ldots)).$$

Additional notation. In extended type symbols, unnecessary parentheses will be omitted. The infixes $\times$ and $+$ will have a smaller scope than $\to$.

As a semantics for these type symbols, we associate with each type symbol α a set D_α:

Definition 1.2 (Type structures) Assume that for each atomic type θ there is a set D_θ. Then we define D_α for each compound type symbol α as follows:

(a) $D_{\alpha\to\beta}$ is the set of all functions with arguments in D_α and values in D_β;

(b) $D_{\alpha\times\beta}$ is the cartesian product $D_\alpha \times D_\beta$ of D_α and D_β; and

(c) $D_{\alpha+\beta}$ is the disjoint sum $D_\alpha + D_\beta$ of D_α and D_β.

A *basic type structure* is then defined to be the set

$$\{D_\alpha | \alpha \text{ is a basic type symbol}\}.$$

An *extended type* structure is defined to be the set

$$\{D_\alpha | \alpha \text{ is an extended type symbol}\}.$$

It is usual in set theory to take for the cartesian product $A \times B$ the set of all ordered pairs $\langle a, b\rangle$ where $a \in A$ and $b \in B$. This is not strictly necessary here: all we really need is an operator $d_{A,B} : A \to B \to A \times B$ and two operators $fst_{A,B} : A \times B \to$ A and $snd_{A,B} : A \times B \to B$ such that $fst_{A,B}(d_{A,B}(a,b)) = a$ and $snd_{A,B}(d_{A,B}(a,b)) = b$. It is not strictly necessary that $d_{A,B}(a,b)$ be the pair $\langle a,b\rangle$, but we will usually think of it that way, and so we will call it a *pairing operator.* The operators $fst_{A,B}$ and $snd_{A,B}$ will be called *projection functions.* If A and B are sets D_α and D_β respectively, then instead of $d_{A,B}$, etc., we shall write $d_{\alpha,\beta}$, etc.

The *disjoint sum* $A + B$ is formed from A and B by making a copy $inl_{A,B}(a)$ of each element $a \in A$ and a copy $inr_{A,B}(b)$ of each $b \in B$ in such a way that each $inl_{A,B}(a)$ is distinct from each $inr_{A,B}(b)$, and then letting $A + B$ be the union of all the copies. In other words,

$$A + B = \{inl_{A,B}(a) | a \in A\} \cup \{inr_{A,B}(b) | b \in B\}.$$

Given any element of this disjoint union, it is possible to tell which of the sets it originally came from. It follows that there is, for any set C, a function

$$case_{A,B,C} : A + B \to (A \to C) \to (B \to C) \to C,$$

such that if $f : A \to C, g : B \to C, a \in A$, and $b \in B$, then

$$case_{A,B,C}(inl_{A,B}(a), f, g) = f(a)$$

and

$$case_{A,B,C}(inr_{A,B}(b), f, g) = g(b).$$

As before, we shall use the notation $case_{\alpha,\beta,\gamma}$ etc.

Often there is an interest in a type which is empty. This type will be called void, and will, for now, be taken as an atomic type. D_{void} will be the empty set.

In some cases, we will want the type N of the natural numbers. This will also be an atomic type, and D_{N} will simply be the set of natural numbers. The successor function will be denoted by σ.

Note that a type structure does not include any set of pairs in which there are pairs in which the first elements are in the same type but the second elements are in different types. Thus, there is no nontrivial way in a type structure to make the type of the second element depend on the first element rather than on the type of the first element. In particular, in a set of pairs whose first elements are natural numbers, all of the second elements must be of the same type. (Of course, sets with pairs whose first elements have the same type but whose second elements have different types can be formed by taking arbitrary unions, but they are not part of a type structure as defined by Definition 1.2.)

1.2 The typed λ-calculus.

So far, we have talked about structures consisting of sets and some functions associated with them. Except for these functions and the natural numbers, we have not talked about any of the elements of the sets. Here, we introduce a formalism of terms which will represent these objects. The formalism we will use is the typed λ-calculus.

The basic idea behind the λ-calculus is the λ-notation of Alonzo Church. The idea is really simple: we are used to saying that if f represents the squaring function, so that $f(x) = x^2$ then $f(2) = 2^2 = 4$. We also sometimes say that this function f is given by $x \mapsto x^2$. We might well ask why we do not write

$$(x \mapsto x^2)(2) = 2^2 = 4.$$

The reason is that in the 1930s, Alonzo Church proposed writing

$$(\lambda x.x^2)(2) = 2^2 = 4. \tag{1.1}$$

This is the basis of the λ-calculus.

In the λ-calculus, we use complete currying. In this notation, the term representing the function h^* of §1 is

$$\lambda x.\lambda y.h(x,y).$$

Since we are interested in terms representing objects in the sets of type structures, we are really interested in the typed λ- calculus. There are a number of forms of this system, depending on which types we are using. Let us begin with the basic type symbols.

Definition 1.3 (Basic typed λ-terms) Assume that we have infinitely many *individual term variables*, where each variable is assigned a type symbol in such a way that there are an infinite number of variables assigned to each type, and suppose that x^α indicates a variable of type (symbol) α. Then *basic typed λ-terms* are defined as follows:

(a) each typed variable x^α is a typed term of type α;

(b) if $M^{\alpha\to\beta}$ and N^α are typed terms of types $\alpha \to \beta$ and α respectively, then $(M^{\alpha\to\beta}N^\alpha)^\beta$ is a typed term of type β; and

(c) if x^α is a variable of type α and $M^\beta$is a term of type β, then $(\lambda x^\alpha.M^\beta)^{\alpha\to\beta}$ is a term of type $\alpha \to \beta$.

A term of the form given by (b) is called an *application term.* A term of the form given by (c) is called an *abstraction term.*

Notation Parentheses will be omitted when no confusion results. For compound application terms, parentheses will be omitted by association to the left, so that

$$M^{\alpha \to \beta \to \gamma \to \delta} N^{\alpha} P^{\beta} Q^{\gamma}$$

is an abbreviation for

$$(((M^{\alpha \to \beta \to \gamma \to \delta} N^{\alpha})^{\beta \to \gamma \to \delta} P^{\beta})^{\gamma \to \delta} Q^{\gamma})^{\delta}$$

Superscripts indicating types will sometimes be omitted when the type is clear from the context.

The notation

$$M \equiv N$$

will mean that "M" and "N" are names for the same term. This notation will be especially used in definitions, such as Definition 1.5 below.

Examples

(a) $(\lambda x^{\alpha}.x^{\alpha})^{\alpha \to \alpha}$ represents the identity function of type α.

(b) If $F^{\beta \to \gamma}$ and $G^{\alpha \to \beta}$ are terms of types $\beta \to \gamma$ and $\alpha \to \beta$ respectively, then $\lambda x^{\alpha}.F^{\beta \to \gamma}(G^{\alpha \to \beta} x^{\alpha})$ represents the composition of the functions represented by $F^{\beta \to \gamma}$ and $G^{\alpha \to \beta}$.

(c) $\lambda x^{\beta \to \gamma}.\lambda y^{\alpha \to \beta}.\lambda z^{\alpha}.x^{\beta \to \gamma}(y^{\alpha \to \beta} z^{\alpha})$, which is a term of type $(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$, represents the operation of composition of functions of types $\alpha \to \beta$ and $\beta \to \gamma$.

(d) If $M^{\alpha}$is a term of type α and $x^{\beta}$is a variable of type β which does not occur free in M^{α}(in the sense of Definition 1.4 below), then $(\lambda x^{\beta}.M^{\alpha})^{\beta \to \alpha}$ represents a constant function whose value for each argument is the object represented by M^{α}.

(e) $\lambda x^{\alpha}.\lambda y^{\beta}.x^{\alpha}$, which is a term of type $\alpha \to \beta \to \gamma$ represents the operator which forms constant functions with arguments in β and value in α.

Definition 1.4 (Free and bound variables) An occurrence of a variable x^{α} in a term M is *bound* if it is in a part of M of the form $\lambda x^{\alpha}.N^{\beta}$; otherwise it is *free*. If x^{α} has at least one free occurrence in M, it is called a *free variable of M*. The set of all free variables of M is called FV(M). A *closed term* is a term without any free variables.

If one of the atomic types is void, then by Definition 1.3 there will be variables of this type. However, it is the intention that there be no *closed* term of type void. A proof that there is no closed term of type void is a kind of consistency result for typed λ-calculus.

Definition 1.5 (Substitution) For a term M^β, a variable x^α, and another term N^α of the same type as the variable, the result of substituting N^α for x^α in M^β, denoted

$$[N^\alpha/x^\alpha]M^\beta,$$

is the result of substituting N^α for each free occurrence of x^α in M^β and changing bound variables to avoid clashes. The precise definition, by induction on the structure of M^β, is as follows, where some type superscripts are omitted:

(a) $[N^\alpha/x^\alpha]x^\alpha \equiv N^\alpha$;

(b) $[N^\alpha/x^\alpha]y^\beta \equiv y^\beta$ for all variables y^β distinct from x^α;

(c) $[N^\alpha/x^\alpha](P^{\gamma\to\beta}Q^\gamma) \equiv ([N^\alpha/x^\alpha]P^{\gamma\to\beta})([N^\alpha/x^\alpha]Q^\gamma)$;

(d) $[N^\alpha/x^\alpha](\lambda x^\alpha.P^\gamma) \equiv \lambda x^\alpha.P^\gamma$;

(e) $[N^\alpha/x^\alpha](\lambda y^\gamma.P^\delta) \equiv \lambda y^\gamma.[N^\alpha/x^\alpha]P^\delta$
if $y^\gamma \not\equiv x^\alpha$ and $y^\gamma \notin \mathrm{FV}(N^\alpha)$ or $x^\alpha \notin \mathrm{FV}(P^\delta)$; and

(f) $[N^\alpha/x^\alpha](\lambda y^\gamma.P^\delta) \equiv \lambda z^\gamma.[N^\alpha/x^\alpha][z^\gamma/y^\gamma]P^\delta$
if $y^\gamma \not\equiv x^\alpha$, $y^\gamma \in \mathrm{FV}(N^\alpha)$, $x^\alpha \in \mathrm{FV}(P^\delta)$, and z^α is the first variable with the same type as y^γ in a standard enumeration of variables which is not in $\mathrm{FV}(N^\alpha)$ or $\mathrm{FV}(P^\delta)$.

If the type of N differs from the type of x, then $[N/x]M$ is not defined.

We are now in a position to introduce a relation which corresponds to the process of calculating values, as in (1.1) above. This relation is called *reduction*. The main idea behind reduction is the instruction we always give beginners for evaluating $f(x)$. For example, if $f(x) = x^2$, the instruction for evaluating $f(2)$ is to replace x by 2, thus getting $2^2 = 4$. This idea gives us the essential relation between a *redex* and its *contractum* in the next definition.

Definition 1.6 (Reduction) A *(one-step) change of bound variable* consists of the replacement of a subterm of a term P^γ of the form

$$\lambda x^\alpha.M^\beta$$

by

$$\lambda y^\alpha.[y^\alpha/x^\alpha]M^\beta,$$

where $y^\alpha \notin \mathrm{FV}(M^\beta)$. A *redex* is a term of the form $(\lambda x^\alpha.M^\beta)N^\alpha$; its *contractum* is $[N^\alpha/x^\alpha]M^\beta$. A *contraction* is the replacement of a redex by its contractum in a term (where the redex before the contraction and the contractum after the contraction are subterms of the term being contracted). A *reduction* is a (possibly empty) sequence of contractions and changes of bound variable.

If M reduces to N, we write

$$M N.$$

Definition 1.7 (Conversion) An *expansion* is the reverse of a contraction; i.e., M expands to N if and only if N contracts to M. A term M is said to *convert to* N if N can be obtained from M by a (possibly empty) sequence of contractions, expansions, and changes of bound variable.

If M converts to N, we write

$$M =_* N.$$

Let us now turn our attention to the other type-forming operators, $\times$ and $+$. For terms of type $\alpha \times \beta$, we need a pairing operator $\mathsf{D}_{\alpha,\beta}$ of type $\alpha \to \beta \to \alpha \times \beta$. We will also want terms representing the projection functions: we want $\mathsf{fst}_{\alpha,\beta}$ and $\mathsf{snd}_{\alpha,\beta}$ of types $\alpha \times \beta \to \alpha$ and $\alpha \times \beta \to \beta$ respectively such that

$$\mathsf{fst}_{\alpha,\beta}(\mathsf{D}_{\alpha,\beta} M^\alpha N^\beta) M^\alpha \text{ and } \mathsf{snd}_{\alpha,\beta}(\mathsf{D}_{\alpha,\beta} M^\alpha N^\beta) N^\beta.$$

To deal with terms of type $\alpha + \beta$, we need terms $\mathsf{inl}_{\alpha,\beta}$, $\mathsf{inr}_{\alpha,\beta}$, and $\mathsf{case}_{\alpha,\beta,\gamma}$ of types $\alpha \to \alpha + \beta$, $\beta \to \alpha + \beta$ and $\alpha + \beta \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$ respectively such that

$$\mathsf{case}_{\alpha,\beta,\gamma}(\mathsf{inl}_{\alpha,\beta} M^\alpha) f^{\alpha\to\gamma} g^{\beta\to\gamma} f^{\alpha\to\gamma} M^\alpha$$

and

$$\mathsf{case}_{\alpha,\beta,\gamma}(\mathsf{inr}_{\alpha,\beta} N^\beta) f^{\alpha\to\gamma} g^{\beta\to\gamma} g^{\beta\to\gamma} N^\beta.$$

We will also want to have natural numbers represented. This can be accomplished by taking one of the atomic type symbols to be N and postulating atomic terms $0^{\mathsf{N}}$of type N, $\sigma^{\mathsf{N}\to\mathsf{N}}$of type $\mathsf{N} \to \mathsf{N}$, and, to represent primitive recursive functions, $\mathsf{R}_\alpha$of type $\alpha \to (\mathsf{N} \to \alpha \to \alpha) \to \mathsf{N} \to \alpha$ such that

$$\mathsf{R}_\alpha M^\alpha N^{\mathsf{N}\to\alpha\to\alpha} 0^{\mathsf{N}} M^\alpha$$

and

$$\mathsf{R}_\alpha M^\alpha N^{\mathsf{N}\to\alpha\to\alpha} (\sigma^{\mathsf{N}\to\mathsf{N}} n^{\mathsf{N}}) N^{\mathsf{N}\to\alpha\to\alpha} n^{\mathsf{N}} (\mathsf{R}_\alpha M^\alpha N^{\mathsf{N}\to\alpha\to\alpha} n^{\mathsf{N}}),$$

where n^{N} is the term representing the natural number n, that is, is the term

$$\sigma^{\mathsf{N}\to\mathsf{N}}(\sigma^{\mathsf{N}\to\mathsf{N}}(\ldots(\sigma^{\mathsf{N}\to\mathsf{N}} 0^{\mathsf{N}})\ldots)), \tag{1.2}$$

where there are n occurrences of $\sigma^{\mathsf{N}\to\mathsf{N}}$.

We are now ready to define extended typed λ-terms.

Definition 1.8 (Extended typed λ-terms) Assume that one of the atomic types is N. Assume that we have individual term variables as in Definition 1.3 and that, in addition, we have the following atomic constants for any types α, β, and γ: $\mathsf{D}_{\alpha,\beta}$ of type $\alpha \to \beta \to \alpha \times \beta$, $\mathsf{fst}_{\alpha,\beta}$ of type $\alpha \times \beta \to \alpha$, $\mathsf{snd}_{\alpha,\beta}$ of type $\alpha \times \beta \to \beta$, $\mathsf{inl}_{\alpha,\beta}$ of type $\alpha \to \alpha + \beta$, $\mathsf{inr}_{\alpha,\beta}$ of type $\beta \to \alpha + \beta$, $\mathsf{case}_{\alpha,\beta,\gamma}$ of type $\alpha + \beta \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$, 0^{N} of type N, $\sigma^{\mathsf{N}\to\mathsf{N}}$ of type $\mathsf{N} \to \mathsf{N}$, and R_α of type $\alpha \to (\mathsf{N} \to \alpha \to \alpha) \to \mathsf{N} \to \alpha$. An *atomic term* is a variable or an atomic constant. *Extended typed* terms are defined as in Definition 3 except that any atomic terms may occur in (a).

Definitions 1.4 and 1.5 hold for extended typed terms as well as for basic typed terms. For reduction, we need some new kinds of redexes. The redexes of Definition 1.6 are called *β-redexes* to distinguish them from the other redexes needed here. (On the significance of this name, see Hindley & Seldin [HS86] Chapter 7)

Definition 1.9 (Reduction) Reduction is defined as in Definition 1.6 except that in addition to β-redexes we now have the following additional redexes (given with their contracta):

	Redex	Contractum
(fst)	$\mathsf{fst}_{\alpha,\beta}(\mathsf{D}_{\alpha,\beta}M^\alpha N^\beta)$	M^α
(snd)	$\mathsf{snd}_{\alpha,\beta}(\mathsf{D}_{\alpha,\beta}M^\alpha N^\beta)$	N^β
(case_1)	$\mathsf{case}_{\alpha,\beta,\gamma}(\mathsf{inl}_{\alpha,\beta}M^\alpha)f^{\alpha\to\gamma}g^{\beta\to\gamma}$	$f^{\alpha\to\gamma}M^\alpha$
(case_2)	$\mathsf{case}_{\alpha,\beta,\gamma}(\mathsf{inr}_{\alpha,\beta}N^\beta)f^{\alpha\to\gamma}g^{\beta\to\gamma}$	$g^{\beta\to\gamma}N^\beta$
(R_1)	$\mathsf{R}_\alpha M^\alpha N^{\mathsf{N}\to\alpha\to\alpha}0^{\mathsf{N}}$	M^α
(R_2)	$\mathsf{R}_\alpha M^\alpha N^{\mathsf{N}\to\alpha\to\alpha}(\sigma^{\mathsf{N}\to\mathsf{N}}n^{\mathsf{N}})$	$N^{\mathsf{N}\to\alpha\to\alpha}n^{\mathsf{N}}(\mathsf{R}_\alpha M^\alpha N^{\mathsf{N}\to\alpha\to\alpha}n^{\mathsf{N}})$

where n^{N} is the term given in (1.2) above.

Definition 1.7 now holds as before.

sectionThe basic theory of typed λ-calculus

Let us begin with the theory of basic typed λ-terms of Definition 1.3.

Lemma 1.1 (Replacement) *If an occurrence of a typed term P^α in a typed term M^β is replaced by another term with type α, then the result is a typed term of type β.*

Proof By induction on the structure of M^β. ■

Theorem 1.1 (Invariance of reduction) *If $M^\alpha N$, then N has type α.*

Proof By Lemma 1.1, it is sufficient to prove that types are preserved by changes of bound variable and that a contractum has the same type as its redex. This will follow in both cases from the fact that $[N^\alpha/x^\alpha]M^\beta$ is a term of type β, and this latter fact can be seen by applying Lemma 1.1 to the cases of Definition1.5. ■

We noted in Section 1.2 above that reduction corresponds to the process of evaluating the result of applying a function to an argument. Since there are many well-known calculations that never come to an end, we might expect to find typed λ-terms that can begin reductions continuing forever. In a trivial sense, most typed λ-terms begin such a reduction, since bound variables can be changed whenever they occur. But changing bound variables does not really correspond to a calculation step; what we really want to know is whether there is a typed terms with the property that every term to which it reduces contains an occurrence of a redex. It turns out that the answer is no.

Definition 1.10 (Normal form) A term is said to be in *normal form* if there is no occurrence of a redex in it. If M^α N^α, where N^α is in normal form, then N^α is said to be *a normal form of M^α*.

Theorem 1.2 (Normal form theorem) *Every basic typed term has a normal form; i.e., every basic typed term can be reduced to a term in normal form.*

Proof Define the *degree* of a type-symbol to be the number of occurrences of the symbol $\rightarrow$ in it, and define the *degree* of a redex $(\lambda x^\alpha.M^\beta)N^\alpha$ to be the degree of the type $\alpha \rightarrow \beta$ of the abstraction part of the redex. The proof is by an induction on the pair $\langle d, n\rangle$, where d is the maximum degree of any redex in the given term and n is the number of occurrences in the term of redexes with degree d. The pairs are ordered by specifying that $\langle d, n\rangle < \langle d', n'\rangle$ if and only if either $d < d'$ or else $d = d'$ and $n < n'$. Since changing bound variables does not change the pair associated

with a given term, it is sufficient to concentrate on the contraction of redexes. At each stage a redex $(\lambda x^\alpha.M^\beta)N^\alpha$ is chosen which has degree d and is such there is no occurrence in N^α of a redex of degree d. The only redexes of degree d in the contractum $[N^\alpha/x^\alpha]M^\beta$ are substitution instances of those occurring in M^β; hence, if the pair associated with the original term is $\langle d, n\rangle$, then the pair associated with the term obtained by carrying out the contraction is $\langle d, n-1\rangle$ if $n > 1$ and is $\langle d', m\rangle$ for $d' < d$ if $n = 1$. (Note that n can never be 0.) Hence, each such contraction leads to a new term with a pair lower in the ordering than the original term, and since the pairs under this ordering are well founded, it follows that the reduction process must terminate in a term in normal form. ■

Corollary 1.2.1 *There is no closed basic typed λ-term in normal form with an atomic type.*

Proof Let P^θ be a closed term in normal form of type θ, where θ is an atomic type. Then P^θ is not a variable, and since θ is atomic, it is not an abstraction term. It follows that P^θ is an application term. Suppose it has the form $P_0P_1 \ldots P_m$, where P_0 is not an application term and type superscripts are omitted for convenience. (Every application term can be written in this form.) If P_0 were an abstraction term, then P^θ would not be in normal form. It follows that P_0 is a variable, and hence P^θ is not a closed term, contrary to hypothesis. ■

This corollary shows that the normalization theorem gives us a kind of consistency result. For if void is one of the atomic types, then it shows that there is no closed term in normal form of type void. Since, as can be easily proved, reduction never introduces any new free variables into a term, it follows that there is no closed term in any atomic type, and hence there is none in void.

There is no problem about extending Lemma 1.1 and Theorem 1.1 to extended typed terms. Furthermore, Theorem 1.2 can be extended to extended typed terms involving (fst), (snd), (case_1), (case_2), and (R_1) redexes. But as soon as (R_2) redexes are allowed, there is a problem, for it is possible to have a subterm of the form $\mathsf{R}_\alpha M^\alpha N^{\mathsf{N}\to\alpha\to\alpha} P^{\mathsf{N}}$ which is not a redex but which becomes a redex after contractions are carried out in P^{N} on redexes of lower degree. However, there is an alternative method of proof, which is more complicated, which proves Theorem 1.1 for extended typed terms with (R_2) redexes. In fact, this stronger method of proof actually proves a stronger result for both the basic and extended systems.

Theorem 1.3 (Strong normalization theorem) *Every sequence of contractions starting with a typed λ-term terminates in a term in normal form.*

For the proof, see Hindley & Seldin [HS86] Appendix 2.

Corollary 1.2.1 is clearly not true in the extended system with terms for the natural numbers, since 0^{N} is a closed term in normal form with atomic type N. However, it is possible to prove that there is no closed term in void. The proof begins like the proof of Corollary 1.2.1, but becomes more complicated at the point of analyzing P_0, for now P_0 might be an atomic constant, and we need a case for each one. For example, we have to consider the possibility that it is $\mathsf{fst}_{\alpha,\beta}$. Furthermore, P_1 has type $\alpha \times \beta$. Since P_1 is in normal form and is closed, it must be of the form $\mathsf{D}_{\alpha,\beta} M^{\alpha} N^{\beta}$, contradicting the assumption that P^{θ} is in normal form. Similar arguments work for the other atomic constants. This proves:

Corollary 1.3.1 *If one of the atomic types is* void, *then there is no closed term of type* void.

We can also obtain a result concerning type N.

Corollary 1.3.2 *Every closed term of type* N *reduces to a numeral; i.e., to a term of the form*

$$\sigma^{\mathsf{N}\to\mathsf{N}}(\sigma^{\mathsf{N}\to\mathsf{N}}(...(\sigma^{\mathsf{N}\to\mathsf{N}} 0^{\mathsf{N}})...)).$$

Proof Given a closed term of type N, let P^{N} be its normal form. The proof is by induction on the structure of the term P^{N}. Follow the proof of Corollary 1.3.1 through the analysis of P_0; there are now additional cases in which it may be 0^{N}, $\sigma^{\mathsf{N}\to\mathsf{N}}$, or R_{α}. If it is 0^{N}, we are done. Otherwise, the second or third argument must be a numeral by the induction hypothesis, and so we either have another numeral or an (R) redex. ■

We would now like to prove that the type structures introduced in section 1 form a model of the extended typed λ- terms.

Definition 1.11 (Valuation) A *valuation* for a given type structure is a function which assigns to each variable $x^{\alpha}$of type α an element $\rho(x^{\alpha})$ of D_{α}. If ρ is a valuation, then $[d/x^{\alpha}]\rho$, where $d \in D_{\alpha}$, is the valuation τ with the property that $\tau(x^{\alpha}) = d$ and, for each variable y^{β}distinct from x^{α}, $\tau(y^{\beta}) = \rho(y^{\beta})$.

Definition 1.12 (Assignment) For each valuation ρ and for each extended typed λ-term M, an object $|M|_{\rho}$, called the assignment of M determined by the valuation ρ, or, when no confusion results, the assignment of M, is defined as follows, where the notation $|M|$ is used when no confusion results:

(a) $|\mathsf{D}_{\alpha,\beta}|$ is the function which, given $d_1 \in D_\alpha$ and $d_2 \in D_\beta$ as arguments, returns the value $d_{\alpha,\beta}(d_1, d_2)$;
(b) $|\mathsf{fst}_{\alpha,\beta}| = fst_{\alpha,\beta} : D_{\alpha\times\beta} \to D_\alpha$;
(c) $|\mathsf{snd}_{\alpha,\beta}| = snd_{\alpha,\beta} : D_{\alpha\times\beta} \to D_\beta$;
(d) $|\mathsf{inl}_{\alpha,\beta}| = inl_{\alpha,\beta} : D_\alpha \to D_{\alpha+\beta}$;
(e) $|\mathsf{inr}_{\alpha,\beta}| = inr_{\alpha,\beta} : Da \to D_{\alpha+\beta}$;
(f) $|\mathsf{case}_{\alpha,\beta,\gamma}| = case_{\alpha,\beta,\gamma} : D_{\alpha+\beta} \to D_{\alpha\to\gamma} \to D_{\beta\to\gamma} \to D_\gamma$;
(g) $|0^{\mathsf{N}}| = 0$;
(h) $|\sigma^{\mathsf{N}\to\mathsf{N}}| = \sigma$;
(i) $|\mathsf{R}_\alpha|$ is the function which, given an element $d \in D_\alpha$ and a function $h : D_{\mathsf{N}} \to D_\alpha \to D_\alpha$, returns as a value the function $f : D_{\mathsf{N}} \to D_\alpha$ with the property that $f(0) = d$ and $f(n+1) = h(n, f(n))$;
(j) $|M^{\alpha\to\beta}N^\alpha| = |M^{\alpha\to\beta}|(|N^\alpha|)$ if this makes sense (i.e., if $|M^{\alpha\to\beta}|$ is a function and $|N^\alpha|$ is an object in its domain);
(k) $|\lambda x^\alpha.M^\beta|_\rho$ is the function $f : D_\alpha \to D_\beta$which, for each element $d \in D_\alpha$, returns $|M^\beta|_\tau$, where τ is $[d/x^\alpha]\rho$.

Theorem 1.4 *For each extended typed λ-term M^α of type α, and for each valuation ρ, $|M^\alpha| \in D_\alpha$. Furthermore, if $M^\alpha =_* N^\alpha$, then $|M^\alpha| = |N^\alpha|$.*

Proof The first part is proved by induction on the structure of M^α. The second part is proved by showing that assignment is invariant of changes of bound variable and that the assignment of any redex is equal to that of its contractum; this follows from Definition 1.12. ■

1.3 The Church-Rosser theorem and pure λ-calculus.

As we have seen, every reduction sequence starting with a typed λ-term terminates in a normal form. But we might well wonder if different reduction sequences terminate in different normal forms. In a trivial sense they do, since a change of bound variable applied to a normal form leads to a distinct normal form. But normal forms which differ only in their bound variables are really essentially the same. What we would like to know is whether or not there are any typed terms which have two or more truly distinct normal forms. The answer turns out to be no: all normal forms of a given typed λ-term differ by only changes of bound variables. This result is a consequence of a theorem due originally to Church & Rosser [CR36].

Theorem 1.5 (Church-Rosser Theorem) *If M, N, and P are typed terms such that PM and PN, then there is a term Q such that MQ and NQ.*

All known proofs of this theorem are too long and complicated to be given here. The most readable proof is probably that of Rosser [Ros84] pp. 342-343. What is perhaps most interesting about this proof (and almost all other published proofs) is that it makes no reference to the type structure; it remains valid if all of the type superscripts are deleted. In fact, the theorem is not really as much a theorem about the *typed* λ-calculus as it is a theorem about the *λ-calculus*. This makes it worth taking a brief look at the *pure λ-calculus*.

Definition 1.13 (Pure λ-terms) Assume that we have infinitely many *variables* and perhaps some *constants*. Then the *(pure) λ-terms* are defined as follows:

(a) Variables and constants are λ-terms;

(b) If M and N are λ-terms, then (MN) is a λ-term; and

(c) If x is a variable and M is a λ-term, then $(\lambda x.M)$ is a λ-term.

Free and bound variables, substitution, reduction, and conversion are defined much as for typed λ-terms; the main difference is that typechecking is not needed in substitution or in forming application terms. Clearly, any typed λ-term can be transformed into a pure λ-term by deleting the type superscripts. On the other hand, there are pure λ-terms to which no typed λ- terms correspond. For example, the term

$$\lambda x.xx$$

does not correspond to any typed term, since there is no typed variable x^α with a type α that permits the formation of $x^\alpha x^\alpha$. Furthermore, the term

$$(\lambda x.xx)(\lambda x.xx)$$

contracts to itself, and so clearly has no normal form. The term

$$(\lambda x.xxx)(\lambda x.xxx)$$

contracts to

$$(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx),$$

and so clearly has no normal form. These last two terms represent computations that do not terminate; the first one represents an infinite loop, and the second represents an expanding infinite loop. Nonterminating computations cannot be represented by typed terms.

The pure λ-calculus differs from the typed λ-calculus in another respect. The typed λ-terms have type structures as models. But the pure λ-calculus does not have such simple models in terms of set theory. The reason for this is that in the pure λ-calculus, any term can be applied to itself: if M is a term, then so is (MM). But the standard axioms of set theory prevent a set-theoretic function (in the usual sense of a set of ordered pairs) from being applied to itself. The typechecking required for the formation of typed application terms is a sufficient restriction to ensure that the terms can be modelled as functions in the ordinary set-theoretic sense.

Chapter 2

EXTENSIONS OF TYPED LAMBDA-CALCULUS

Although the typed λ-calculus, which we saw in Chapter 1, is in an important sense the basis of the theory of constructions, the theory of constructions is not exactly a form of typed λ-calculus; it is actually a form of deductive system for assigning types to λ-terms. There are a number of such deductive systems, and we will look at a several of them in this chapter. The ones at which we will look will approximate a sequence of systems leading from the weakest, basic type assignment, to the strongest, which is the theory of constructions itself.

We begin with a basic system of type assignment, TA, which is equivalent to the ordinary typed λ-calculus. This system is much weaker than the theory of constructions, but its theory illustrates very well what we will want later for the theory of constructions itself. This system and its theory are considered in the first two sections. We then proceed, in the next two sections, to consider the second order polymorhpic typed λ-calculus, which is one of the best known generalizations of ordinary type assignment and is of considerable interest to computer scientists in connection with polymorphism in programming languages. We will see some of the strength of this system.

The theory of constructions is a form of what is usually called *generalized type assignment*, which we will consider in the last four sections of the chaper. We begin first with a general description of the sort of generalization that is involved (Section 2.5), and we then see (Section 2.6) why systems of this sort require conversion on the types. We look at the basic system of generalized type assignment in Section 2.7, and we see that it is, in a sense, a conservative extension of ordinary type assignment. Finally, in Section 2.8, we look at some stronger systems that point

the way to the theory of constructions; the most important of these is the universal fragment of the type theory of Martin-Löf, but, as we shall see, this system is not even strong enough to interpret the second order polymorphic typed λ-calculus, and we look at how the former sysem would have to be strengthened to interpret the latter. We end with some limitations on the system which results from this strengthening and which are overcome in the theory of constructions itself.

It is worth mentioning that it is desirable to interpret the second order polymorphic typed λ-calculus in systems of generalized type assignment because of the strength of the former, which we will see in Section 2.4, and the fact that we have a method for proving the consistency of the latter. In general, when we have a system which can be proved consistent and in which we can interpret other systems, the latter systems are shown to be consistent. As we shall see in Chapter 5, the consistency proof for the theory of constructions leads to consistency results for the interpretations of a number of useful theories from mathematics and logic.

2.1 Type assignment

In the typed λ-calculus as defined above, terms without types cannot be formed. But in most programming languages with type discipline, types play a different role: instead of preventing terms from being formed, they pick out of a set of terms that already exist those terms that are acceptable to a programming context (such as a compiler). The terms exist independently of the types, and the relationship between the types and the terms is established by a process of assigning types to terms.

It turns out to be easy to apply this approach to the λ-calculus. We need only assume that we are dealing with the pure λ-terms of Definition 1.13 and give a systemmatic procedure for assigning types to them.

This procedure will take the form of a deductive theory or system. The formulas of the system will all have the form

$$M : \alpha,$$

where M is a term and α is a type. The *axioms* will be formulas assigning types to the atomic constants if there are any. (For the moment, let us make things simpler by assuming that there are no atomic constants.) We also need to assign types to the variables. In the definition of basic typed terms (Definition 1.3), we postulated that each variable came with a type. Here, we do not postulate this. Instead, we will postulate that in any particular assignment, types are assigned to the variables by assumption. In general, Γ will be a set of such assumptions; i.e., Γ will be a set of formulas of the form

$$x_1 : \alpha_1,\ x_2 : \alpha_2,\ \ldots,\ x_n : \alpha_n,$$

where $x_1, x_2, \ldots, x_n$ are distinct variables and $\alpha_1, \alpha_2, \ldots, \alpha_n$ are types. Thus, in general, an assignment of a type to a term is a deduction whose assumptions assign types to the free variables in the term. The statement that $M : \alpha$ can be deduced from a set of assumptions Γ will be written

$$\Gamma \vdash M : \alpha.$$

If we look at the definition of pure λ-terms, we will see that we have taken care of assigning types to the atomic terms (constants and variables). To assign types to compound terms, we need rules. These rules will have to correspond to the clauses assigning types to application terms and abstraction terms in the definition of basic typed λ-terms, Definition 1.3. They are as follows:

($\rightarrow$ e) If $\Gamma \vdash M : \alpha \rightarrow \beta$ and $\Gamma \vdash N : \alpha$, then $\Gamma \vdash (MN) : \beta$.

($\rightarrow$ i) If $\Gamma,\ x:\alpha \vdash M:\beta$, where x does not occur free in Γ, then $\Gamma \vdash \lambda x.M:\alpha\rightarrow\beta$.

Note in the case of ($\rightarrow$ i), the conclusion of the rule does not depend on the assumption $x:\alpha$, whereas the premise does. We say that the assumption is discharged by the rule. This notion of discharging an assumption is quite common in natural deduction formulations of systems of logic, which were introduced originally by Jaśkowski [Jas34] and Gentzen [Gen34] and were extensively studied by Prawitz [Pra65] . In these systems, the above rules would usually be written as follows:

$$(\rightarrow \text{e})\quad \frac{M:\alpha\rightarrow\beta \qquad N:\alpha}{MN:\beta} \qquad\qquad (\rightarrow \text{i})\quad \frac{\begin{array}{c}[x:\alpha]\\ M:\beta\end{array}}{\lambda x.M:\alpha\rightarrow\beta,}$$

where in ($\rightarrow$ i), x does not occur free in any undischarged assumption, and where the square brackets indicate the discharging of the assumption $x:\alpha$ by the rule.

Writing the rules this way is associated with writing deductions as trees, as the following examples indicate:

Example 2.1 $\lambda x.x:\alpha\rightarrow\alpha$ for each type α.

Proof

$$\frac{\begin{array}{c}1\\ [x:\alpha]\end{array}}{\lambda x.x:\alpha\rightarrow\alpha}\ (\rightarrow \text{i}-1)$$

■

Here the brackets indicate the discharged assumption, and the number "1" is used to indicate the location of the discharge. The importance of keeping track of the places at which assumptions are discharged is shown in the following example:

Example 2.2 For any types α, β, and γ, we have

$$\lambda x.\lambda y.\lambda z.xz(yz):(\alpha\rightarrow\beta\rightarrow\gamma)\rightarrow(\alpha\rightarrow\beta)\rightarrow\alpha\rightarrow\gamma.$$

Proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{3}{[x:\alpha\to\beta\to\gamma]}\quad\dfrac{1}{[z:\alpha]}}{xz:\beta\to\gamma}(\to e)\quad\dfrac{\dfrac{2}{[y:\alpha\to\beta]}\quad\dfrac{1}{[z:\alpha]}}{yz:\beta}(\to e)}{xz(yz):\gamma}(\to e)}{\lambda z.xz(yz):\alpha\to\gamma}(\to i-1)}{\lambda y.\lambda z.xz(yz):(\alpha\to\beta)\to\alpha\to\gamma}(\to i-2)}{\lambda x.\lambda y.\lambda z.xz(yz):(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma.}(\to i-3)$$

■

It is important to note that an assumption which is discharged need not actually be used. Consider the following example:

Example 2.3 For any types α and β, $\lambda x.\lambda y.x : \alpha\to\beta\to\alpha$.

Proof

$$\dfrac{\dfrac{\dfrac{1}{[x:\alpha]}}{\lambda y.x:\beta\to\alpha}(\to i-v)}{\lambda x.\lambda y.x:\alpha\to\beta\to\alpha}(\to i-1)$$

■

Here, the assumption discharged at the first step is $y:\beta$, which does not actually appear in the deduction. The "- v" indicates this fact.

This method of writing deductions and proofs is common in logic and is appropriate for theoretical purposes, as we shall see. But many non-logicians may be uncomfortable with writing deductions as trees. An alternative is to write the deductions as tables. The three examples given above can be written as follows:

	Formula	Rule	Assumptions
	Example 2.1′		
1.	$x : \alpha$	Hyp	1
2.	$\lambda x.x : \alpha \to \alpha$	1 ($\to$ i)	
	Example 2.2′		
1.	$x : \alpha \to \beta \to \gamma$	Hyp	1
2.	$y : \alpha \to \beta$	Hyp	2
3.	$z : \alpha$	Hyp	3
4.	$xz : \beta \to \gamma$	1,3 ($\to$ e)	1,3
5.	$yz : \beta$	2,3 ($\to$ e)	2,3
6.	$xz(yz) : \gamma$	4,5 ($\to$ e)	1,2,3
7.	$\lambda z.xz(yz) : \alpha \to \gamma$	6 ($\to$ i)	1,2
8.	$\lambda y.\lambda z.xz(yz) : (\alpha \to \beta) \to \alpha \to \gamma$	7 ($\to$ i)	1
9.	$\lambda x.\lambda y.\lambda z.xz(yz) :$ $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$	8 ($\to$ i)	
	Example 2.3′		
1.	$x : \alpha$	Hyp	1
2.	$\lambda y.x : \beta \to \alpha$	1 ($\to$ i)	1
3.	$\lambda x.\lambda y.x : \alpha \to \beta \to \alpha$	2 ($\to$ i)	

Note that here the discharge of an assumption is indicated by the removal of its number from the last column, and that if ($\to$ i) is used without a change in the last column, then the discharge is vacuous.

One feature of this kind of system is that these proofs can all be obtained by working backwards. Let us see this for each of the three examples:

Example 2.1″ We want to prove

$$\vdash \lambda x.x : \alpha \to \alpha.$$

The only rule of which this can be the conclusion is ($\to$ i), and the premise must be

$$x : \alpha \vdash x : \alpha.$$

But this is a trivial deduction consisting of an assumption. ■

Example 2.2″ We want to prove

$$\vdash \lambda x.\lambda y.\lambda z.xz(yz) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma.$$

This must be the conclusion of ($\to$ i), and the premise must be

$$x : \alpha \to \beta \to \gamma \vdash \lambda y.\lambda z.xz(yz) : (\alpha \to \beta) \to \alpha \to \gamma.$$

This must also be the conclusion of ($\to$ i) with the premise

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta \vdash \lambda y.\lambda z.xz(yz) : \alpha \to \gamma.$$

This must also be the conclusion of ($\to$ i), and the premise must be

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash xz(yz) : \gamma.$$

Now this must be the conclusion of ($\to$ e), and the premises must be

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash xz : \delta \to \gamma \tag{2.1}$$

and

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash yz : \delta \tag{2.2}$$

for some type δ. Now each of these must also be the conclusion of an inference by ($\to$ e). The premises for (2.1) must be

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash x : \epsilon \to \delta \to \gamma$$

and

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash z : \epsilon$$

for some type ϵ, and it is clear that these deductions are trivial if δ is β and ϵ is α. Then (2.2) must be

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash yz : \beta,$$

and its premises must be

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash y : \zeta \to \beta$$

and

$$x : \alpha \to \beta \to \gamma,\ y : \alpha \to \beta,\ z : \alpha \vdash z : \zeta.$$

These two deductions also become trivial if ζ is α. ■

Example 2.3″ We need to prove

$$\vdash \lambda x.\lambda y.x : \alpha \to \beta \to \alpha.$$

This must be the conclusion of an inference by ($\to$ i), and the premise must be

$$x : \alpha \vdash \lambda y.x : \beta \to \alpha.$$

This must also be the conclusion of an inference by ($\to$ i), and the premise must be

$$x : \alpha,\ y : \beta \vdash x : \alpha,$$

which is a trivial deduction. ■

This style of finding deductions is called the *refinement style*, and is close to the usual method of implementing on a computer procedures for constructing proofs in this kind of system.

Let us give this system a name. Note that for technical reasons, we need one additional rule which has not been needed in the above examples.

Definition 2.1 (The type-assignment system TA) The system TA is a natural deduction system. Its formulas, called type- assignment formulas, are the expressions of the form

$$M : \alpha,$$

where M is a pure term and α is a (basic) type symbol. There are no axioms. The rules are as follows:

$$(\to \text{e}) \quad \frac{M : \alpha \to \beta \qquad N : \alpha}{MN : \beta}$$

$$(\to \text{i}) \quad \frac{\begin{array}{c}[x : \alpha] \\ M : \beta\end{array}}{\lambda x.M : \alpha \to \beta}$$

Condition: $x : \alpha$ is the only undischarged assumption in which x occurs free.

$(\equiv_\alpha)$ $\dfrac{M : \beta}{N : \beta}$ *Condition:* N is obtained from M by change of bound variables and $M : \beta$ is not the conclusion of a rule.

Note that rule $(\equiv_\alpha)$ cannot occur in a deduction if all assumptions are of the form $x : \alpha$, where x is a variable. The rule is included to allow assumptions of other forms and because we will need it in systems we will take up later.

There are several things to note about this system. The first is that deductions invariably follow the construction of the term to which a type is assigned by the conclusion. This fact, which is easy to see, is difficult to write out as a formal theorem. It is known as the *subject-construction theorem*; see Curry, Hindley & Seldin [CHS72] Theorem 14D1, p. 310. (The name comes from the fact that the term M in a formula $M : \alpha$ is called the subject of the formula.) Nevertheless, it should be obvious from the above examples. One result of this theorem is that it is fairly easy to determine the type of any bound variable. Another is that it is decidable whether or not a given term has a type. See the discussion in Hindley & Seldin [HS86] Chapter 15.

By using the subject-construction theorem, we can obtain results for deductions in TA corresponding to the results of Section 1.3 above for basic terms. First, we need to define a basis as a set of assumptions of the form

$$M_1 : \alpha_1, \ldots, M_n : \alpha_n.$$

A variables-only basis is a basis in which each M_i is a variable. Then, we have the following analogue of Lemma 1.1:

Lemma 2.1 (Replacement) *Let Γ_1 be any basis, and let $\mathcal{D}$ be a deduction giving*

$$\Gamma_1 \vdash_{\mathrm{TA}} M : \alpha.$$

Let P be a term occurrence in M, and let λx_1 , ..., λx_n be those λ's whose scope contains P. Let $\mathcal{D}$ contain a formula $P : \gamma$ in the same position that P has in the construction tree of M, and let

$$x_1 : \delta_1, \ldots, x_n : \delta_n$$

be the assumptions above $P : \gamma$ *that are discharged by applications of* $(\rightarrow \text{i})$ *below it. Assume that* $P : \gamma$ *is not in* Γ_1 *. Let* Q *be a term such that* $\text{FV}(Q) \subset \text{FV}(P)$*, and let* Γ_2 *be a basis in which* $x_1, \ldots, x_n$ *do not occur free such that*

$$\Gamma_2,\ x_1 : \delta_1, \ldots, x_n : \delta_n \vdash_{\text{TA}} Q : \gamma.$$

Let M^* *be the result of replacing* P *by* Q *in* M*. Then*

$$\Gamma_1 \cup \Gamma_2 \vdash_{\text{TA}} M^* : \alpha.$$

Proof See Hindley & Seldin [HS86] Lemma 15.16. ■

Using this lemma and the subject-construction theorem, it is easy to prove the following theorem:

Theorem 2.1 (Subject-reduction theorem) *Let* Γ *be a variables-only basis. If*

$$\Gamma \vdash_{\text{TA}} M : \alpha$$

and MN*, then*

$$\Gamma \vdash_{\text{TA}} N : \alpha.$$

Proof See Hindley & Seldin [HS86] Theorem 15.17. ■

From these results, we can see that deductions in TA correspond to typed terms in the sense of Definition 1.3.

Definition 2.2 (Correspondence between deductions and terms) For each deduction $\mathcal{D}$ of TA, a typed term $|\mathcal{D}|$ in the sense of Definition 1.3 whose type is the type of the conclusion of $\mathcal{D}$, is defined as follows:
(a) If $M : \alpha$ is an assumption, then $|M : \alpha|$ is a typed variable x^α of type α. This variable must be so chosen that it is not assigned to any other assumption which is not also of the form Mca ; but if $M : \alpha$ is a discharged assumption then the same variable must be assigned to any other assumptions of the form $M : \alpha$ which are discharged at the same inference by $(\rightarrow \text{i})$;
(b) If $\mathcal{D}$ is

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ M : \alpha \rightarrow \beta & N : \alpha\end{array}}{MN : \beta}\ (\rightarrow \text{e})$$

then $|\mathcal{D}| \equiv |\mathcal{D}_1||\mathcal{D}_2|$;
(c) If $\mathcal{D}$ is

$$\dfrac{\begin{array}{c}1\\ [x:\alpha]\\ \mathcal{D}_1\\ M:\beta\end{array}}{\lambda x.M:\alpha\to\beta}\ (\to \text{i}-1)$$

then $|\mathcal{D}| \equiv \lambda v^{\alpha}.|\mathcal{D}_1|$ where $v^{\alpha} \equiv |x:\alpha|$.

(This is not quite a one-to-one correspondence because the condition on typed variables in (a) is almost impossible to satisfy with one definition for all deductions in a way that is consistent with the changes of bound variables required to define substitution. But for any small set of deductions, it is locally a one-to-one correspondence.)

This correspondence suggests that we define reduction steps for deductions as well as for terms. These reduction steps turn out to be similar to the $\supset$-reduction steps of Prawitz [Pra65] (see Section 3.3):

Definition 2.3 (β-reduction steps for deductions) A deduction of the form

$$\dfrac{\dfrac{\begin{array}{c}1\\ [x:\alpha]\\ \mathcal{D}_1(x)\\ M:\beta\end{array}}{\lambda x.M:\alpha\to\beta}\ (\to \text{i}-1) \qquad \begin{array}{c}\mathcal{D}_2\\ N:\alpha\end{array}}{(\lambda x.M)N:\beta}\ (\to \text{e})$$

$$\mathcal{D}_3$$

reduces to

$$\mathcal{D}_2$$
$$N : \alpha$$
$$\mathcal{D}_1(N)$$
$$[N/x]M : \beta$$
$$\mathcal{D}_3'$$

where $\mathcal{D}_3'$ is obtained from $\mathcal{D}_3$ by replacing appropriate occurrences of $(\lambda x.M)N$ by $[N/x]M$ according to Lemma 2.1.

Using Definition 2.3 , we can prove the following result:

Theorem 2.2 (Normalization theorem for deductions) *Every deduction in TA can be reduced to a deduction which cannot be reduced further.*

This can also be proved directly; see Hindley & Seldin [HS86] Theorem 15.31.

By the subject-construction theorem, it follows that if there is a deduction $\mathcal{D}$ of $M : \alpha$ from a variables-only basis, and if there is a β-redex in M, then $\mathcal{D}$ can be reduced by a β-reduction step for deductions. This gives us the following corollary.

Corollary 2.2.1 (Normalization theorem for terms) *Let* Γ *be a variables only basis. If*

$$\Gamma \vdash_{\mathrm{TA}} M : \alpha,$$

then M *has a normal form.*

(See Hindley & Seldin [HS86] Corollary 15.31.1.)

A deduction which cannot be further reduced, which is usually called a normal deduction, has the property that there is no inference by ($\rightarrow$ i) whose conclusion is the major (left) premise for an inference by ($\rightarrow$ e). It follows from this that if one takes a normal deduction (in tree form) and starts with any assumption, whether discharged or not, then, as one proceeds down the tree, one cannot come to a major premise for an inference by ($\rightarrow$ e) below an inference by ($\rightarrow$ i) unless one passes through a minor (right) premise for an inference by ($\rightarrow$ e) in between. Let us define a branch of a deduction to be a sequence $A_1, A_2, \ldots A_n$ of formula occurrences such that A_1 is a (discharged or undischarged) assumption, for each $i < n$, A_i is the premise for an inference (but not the right premise for an inference by ($\rightarrow$ e)) and A_{i+1} is the conclusion, and A_n is either the conclusion of the deduction or else

the right premise for an inference by ($\rightarrow$ e). Then each branch consists of zero or more left premises for inferences by ($\rightarrow$ e) followed by premises for inferences by ($\rightarrow$ i). (Under certain circumstances, a branch may begin with the premise for an inference by ($\equiv_\alpha$).) It follows that any deduction proceeds by breaking the types of the assumptions down into their constituent parts and then putting the parts back together to get the type of the conclusion. There are a number of consequences of this fact, among them the following:

Corollary 2.2.2 (Subtype property) *In any normal deduction in TA, every type appearing in a formula of the deduction is a subtype of the type of one of the assumptions or else of the conclusion.*

Another consequence of this structure of normal deductions is the following:

Corollary 2.2.3 *If the type of the conclusion of a normal deduction is atomic, then there is no inference by* ($\rightarrow$ i) *in the leftmost branch* (*i.e., the branch that begins with the top left assumption and ends with the conclusion of the deduction*).

Remark It is not hard to extend this theory to extended typed λ-terms. All we need to do is to add some new constants and assign them new types using axiom schemes as follows:

(D) $\mathsf{D}_{\alpha,\beta} : \alpha \rightarrow \beta \rightarrow \alpha \times \beta$,
(fst) $\mathsf{fst}_{\alpha,\beta} : \alpha \times \beta \rightarrow \alpha$,
(snd) $\mathsf{snd}_{\alpha,\beta} : \alpha \times \beta \rightarrow \beta$,
(inl) $\mathsf{inl}_{\alpha,\beta} : \alpha \rightarrow \alpha + \beta$,
(inr) $\mathsf{inr}_{\alpha,\beta} : \beta \rightarrow \alpha + \beta$,
(case) $\mathsf{case}_{\alpha,\beta,\gamma} : \alpha + \beta \rightarrow (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \gamma$,
(0) $\mathsf{0} : \mathsf{N}$,
(σ) $\sigma : \mathsf{N} \rightarrow \mathsf{N}$,
and
(R_α) $\mathsf{R}_\alpha : \alpha \rightarrow (\mathsf{N} \rightarrow \alpha \rightarrow \alpha) \rightarrow \mathsf{N} \rightarrow \alpha$.

We also assume that these constants satisfy the contractions obtained from the first four of Definition 1.9 by dropping type superscripts. For some purposes, as we shall see in Section 3.4, we are not interested in the constants 0, σ, and R_α. The system without the constants 0, σ, and R_α(and without the atomic type N) will be called *extended* TA. The system with N, 0, σ, and $\mathsf{R}_\alpha$will be called *extended* TA *with arithmetic.*

2.2 Type variables and principal type scheme

As we saw in Example 2.1 above,

$$\lambda x.x : \alpha \to \alpha$$

for every type α. It follows that if θ is any atomic type, then

$$\lambda x.x : \theta \to \theta.$$

It seems clear that any other type assigned to $\lambda x.x$ can be obtained from the type $\theta \to \theta$ by "substituting" some other type for θ. It would be nice to formalize and generalize this property of type-assignment.

The notion of "substitution" into a type would make more sense if we had type variables. Hence, we extend Definition 2.1 as follows:

Definition 2.4 (Type schemes) The *atomic type constants* or *type constants* will be the atomic type symbols of Definition 1.1. We assume that we have infinitely many *type variables*, which will be denoted a, b, etc. Then *type schemes* are defined as follows:

(a) Type constants and type variables are (atomic) type schemes;

(b) If α and β are type schemes, then so is $(\alpha \to \beta)$.

A *type* is a type scheme in which no type variables occur. A type scheme β is a *substitution instance* of a type scheme α if β is obtained from α by substituting types for type variables; i.e., if there are type variables $a_1, a_2, \ldots, a_n$ and type schemes $\gamma_1, \gamma_2, \ldots, \gamma_n$ such that

$$\beta \equiv [\gamma_1/a_1, \gamma_2/a_2, \ldots, \gamma_n/a_n]\alpha.^1$$

From now on, we will assume that TA is defined using type schemes instead of types.

Now the property of type assignment that we noted at the beginning of this section can be formulated by saying that any type or type scheme assigned to $\lambda x.x$ is a substitution instance of $a \to a$. We are interested in knowing which terms are assigned a type scheme with the property that any other type scheme assigned to the term is a substitution instance of the given one. A type scheme with this property deserves a special name.

[1] We are ignoring for the moment types $\alpha \times \beta$ and $\alpha + \beta$. The reasons for this will become apparent in Section 2.4 below.

Definition 2.5 (Principal type scheme) Let M be a closed term. Then a type scheme α is called a *principal type scheme* (*p.t.s.*) of M if and only if

$$\vdash_{\mathrm{TA}} M : \alpha'$$

holds for a type scheme α' when and only when α' is a substitution instance of α.

This definition clearly works only for closed terms; i.e., for terms with no free variables. For terms with free variables, we need to generalize this definition. First, we define an FV(M)-*basis* for a term M to be a basis of the form

$$M_1 : \alpha_1,\ M_2 : \alpha_2,\ \ldots,\ M_n : \alpha_n,$$

in which each M_i is a variable which occurs free in M.

Definition 2.6 (Principal pair) Let M be a term whose free variables are $x_1, x_2, \ldots, x_n$. Then a pair $\langle \Gamma, \alpha \rangle$ is called a *principal pair* (*p.p.*) of M, and α a *p.t.s.* of M, if and only if Γ is an FV(M)-basis and

$$\Gamma' \vdash_{\mathrm{TA}} M : \alpha'$$

holds for an FV(M)-basis Γ' and a type scheme α' when and only when Γ' and α' are obtained from Γ and α respectively by the same *substitution*.

Example 2.4 $\lambda x.x$ has p.t.s. $a \to a$.

Example 2.5 $\lambda x.xx$ is not assigned any type by TA.

These examples should make it clear that the following theorem holds; its proof, although simple in principle, is complicated to write out and will not be given here. (See Hindley & Seldin [HS86] Theorem 15.26 and Theorem 14.40.)

Theorem 2.3 (P.t.s. theorem) *Every pure λ-term M to which a type scheme is assigned by* TA *using only* FV(M)*-bases has a p.t.s. and a p.p.*

It is worth noting that the use of type variables makes it possible to make general assertions. The fact that $\lambda x.x$ has as a p.t.s. $a \to a$ means that it has type $\alpha \to \alpha$ for all types α. Thus, a statement such as

$$\vdash_{\mathrm{TA}} \lambda x.x : a \to a$$

makes a statement about all types α. This same method of making general statements about types is used in the programming language ML (see Gordon et al. [GMW79] and Milner [Mil85] and [Mil78]).

2.3 Universal quantification over all types

We have seen how to use type variables to make statements about all types. But the system we have above is still not what is usually needed for making and using such statements in a programming language. For example, in a language such as FORTRAN or PASCAL, programs that differ only in the types of their variables need to be duplicated and compiled separately. A language such as ML avoids this problem by using type variables and having a rule of substitution for them. We could easily imitate ML by adding a rule such as

$$\frac{M : \alpha}{M : [\beta/a]\alpha} ,$$

but this seems to be in some ways incompatible with the subject-construction theorem. The alternative which suggests itself is to add an explicit universal quantifier.

A system with this explicit universal quantifier is already known; it was introduced independently by Girard [Gir71] and Reynolds [Rey74]. The definition of type is extended by specifying that if a is a type variable and α is a type, then $(\forall a)\alpha$ is a type. For this to make complete sense, we need to keep track of the types of bound variables; thus, if the type of x is α, then we shall write $\lambda x{:}\alpha \,.\, M$ instead of $\lambda x.M$. For example, the identity function on type α will now be written $\lambda x{:}\alpha \,.\, x$. If we take the type to be the type variable a, then we have $\lambda x{:}a \,.\, x$, which has type $a \to a$. Obviously, some term related to this one should be in the type $(\forall a)(a \to a)$, and the fact that the term has this type should express the fact that in TA a p.t.s. of $\lambda x.x$ is $a \to a$. To construct the term we need, we add a new abstraction operator, from a type variable a and a term M. In our example, the term in $(\forall a)(a \to a)$ is $\lambda a \,.\, \lambda x{:}a \,.\, x$. To go with this new abstraction operator, we need a new application: the result of applying a term M to a type-scheme β will be $M\beta$. In our example, we will have the term $(\lambda a \,.\, \lambda x{:}a \,.\, x)\beta$, which we expect to be assigned type $\beta \to \beta$ and to reduce to $\lambda x{:}\beta \,.\, x$. In general, we expect to have the "β"-contraction of $(\lambda a.M)\beta$ to $[\beta/a]M$. We also have the following new type assignment rules:

(∀e) $\dfrac{M : (\forall a)\alpha}{M\beta : [\beta/a]\alpha}$ *Condition:* β is a type.

(∀i) $\dfrac{M : \alpha}{\lambda a.M : (\forall a)\alpha}$ *Condition:* a does not occur free in any undischarged assumption.

One effect of these rules is to give us functions which take types as arguments. Such functions cannot be represented in the type structures of Section 2.1. See the second note before Example 2.6 below.

Note that with our new notation, rule ($\rightarrow$ i) is now written as follows:

$$\frac{\begin{array}{c}1\\ [x:\alpha]\\ M:\beta\end{array}}{\lambda x{:}\alpha\,.\,M:\alpha\rightarrow\beta.}$$

The system defined this way is called the second-order polymorphic typed λ-calculus, or, for short, second-order λ-calculus. To define it, we have the following formal definitions:

Definition 2.7 (Second-order polymorphic types and type schemes) Assume that we have some *type constants* and infinitely many *type variables* as in Definition 2.4. Then *second-order polymorphic type schemes* are defined as follows:

(a) all type constants and type variables are type schemes;

(b) if α and β are type schemes, then so is $(\alpha\rightarrow\beta)$; and

(c) if α is a type scheme and a is a type variable, then $(\forall a)\alpha$ is a type scheme. An occurrence of a type variable a in a type scheme α is said to be *bound* if it is inside a subtype scheme of the form $(\forall a)\alpha$; otherwise it is *free.* A *second-order polymorphic type* is a second-order polymorphic type scheme in which every occurrence of a type variable is bound. The set of all type variables free in α is called $\mathrm{FV}(\alpha)$.

Definition 2.8 (Second-order polymorphic λ-terms) Assume that we have infinitely many *term variables*, distinct from the type variables, and perhaps some *constants*, each constant having a type scheme assigned to it. Then *second-order polymorphic λ-terms* are defined as follows:

(a) every constant and variable is a term;

(b) if M and N are terms, then so is (MN);

(c) if x is a variable, α a type scheme, and M a term, then $(\lambda x{:}\alpha\,.\,M)$ is a term;

(d) if M is a term and α is a type scheme, then $M\alpha$ is a term; and

(e) if a is a type variable and M is a term, then $(\lambda a.M)$ is a term.

An occurrence of a term variable x in a term P is said to be *bound* if it is inside a

subterm of the form $\lambda x{:}\alpha\ .\ M$; otherwise it is *free.* An occurrence of a type variable a in a term P is *bound* if it is inside a subterm of the form $\lambda a.M$; otherwise it is *free.* The set of all term and type variables free in M is called $\mathrm{FV}(M)$.

Definition 2.9 (Substitution) Substitution of terms for term variables and type schemes for type variables is defined much as in Definition 2.6; in particular, bound term and type variables are automatically changed to avoid conflicts.

Definition 2.10 (Change of bound variables) A *change of bound variables* in a type scheme or term is any of the following replacements:
(a) $(\forall a)\beta$ by $(\forall b)[b/a]\beta$ if $b \notin \mathrm{FV}(\beta)$;
(b) $\lambda a.M$ by $\lambda b.[b/a]M$ if $b \notin \mathrm{FV}(M)$;
(c) $\lambda x{:}\beta\ .\ M$ by $\lambda y{:}\beta\ .\ [y/x]M$ if $y \notin \mathrm{FV}(M)$.

Definition 2.11 (β-reduction) For terms P and Q, we say that P *β-reduces* to Q ($P \ _\beta\ Q$, or PQ) if and only if Q is obtained from P by a finite (perhaps empty) series of changes of bound variables and the following kinds of contractions:
(β^1) $(\lambda x{:}\alpha\ .\ M)N \ _\beta\ [N/x]M$;
(β^2) $(\lambda a.M)\alpha \ _\beta\ [\alpha/a]M$.

Conversion is defined from this reduction as in Definition 1.7.

Definition 2.12 (The type assignment system TAP)
TAP (second-order polymorphic type assignment) is a natural deduction system. Its formulas are the type assignment formulas

$$M : \alpha,$$

where M is a second-order polymorphic term (Definition 2.8) and α is a second-order polymorphic type scheme (Definition 2.7). TAP has axioms which assign types to atomic constants if there are any; otherwise it has no axioms. Its rules are

as follows:

$$(\to \text{e}) \quad \frac{M : \alpha \to \beta \qquad N : \alpha}{MN : \beta}$$

$$(\to \text{i}) \quad \frac{\begin{array}{c}[x : \alpha] \\ M : \beta\end{array}}{\lambda x{:}\alpha \,.\, M : \alpha \to \beta}$$

Condition: x is a term variable which is not free in any undischarged assumption.

$$(\forall \text{e}) \quad \frac{M : (\forall a)\alpha}{M\beta : [\beta/a]\alpha}$$

Condition: β is a type scheme.

$$(\forall \text{i}) \quad \frac{M : \alpha}{\lambda a.M : (\forall a)\alpha}$$

Condition: a is a type variable which is not free in any undischarged assumption.

$$(\equiv'_\alpha) \quad \frac{M : \beta}{N : \beta}$$

Condition: N is obtained from M by changes of bound variables.

$$(\equiv''_\alpha) \quad \frac{M : \beta}{M : \gamma}$$

Condition: γ is obtained from β by changes of bound variables and $M : \beta$ is not the conclusion of a rule.

Notes

1. Rules $(\equiv'_\alpha)$ and $(\equiv''_\alpha)$ have not been postulated in the literature; however, it is standard to ignore changes of bound variables and the rules seem necessary to formalize this practice. Note that while rule $(\equiv''_\alpha)$ is restricted the way rule $(\equiv_\alpha)$ is in TA (Definition 2.1), rule $(\equiv'_\alpha)$ is not. In fact, if the latter rule were so restricted, it would be impossible to deduce statements of the form $\lambda a.M$:

$(\forall b)\beta$ unless a and b were the same or there were an assumption of this form.

2. As we saw above we now have functions which take types for arguments, which are not part of the type structures defined in Section 2.1, so these type structures are not models for TAP. In fact, Reynolds [Rey84] has shown that there are no models for TAP in which the types are interpreted as sets as in type structures. There are models of TAP in terms of category theory, but many people who do not know category theory do not find such models helpful. For computer scientists, it is probably best to think of the terms of TAP as having only computational meaning.

3. Some writers use a different notation: $M\{\alpha\}$ instead of $M\alpha$ and $\Lambda a.M$ for $\lambda a.M$. The notation used here does not hide any important distinctions which are not clear from the context and is somewhat cleaner than the alternative.

Example 2.6 The informal discussion before Definition 2.7 corresponds to the following formal deduction in TAP:

$$\dfrac{\dfrac{\dfrac{\dfrac{1}{[x:a]}}{\lambda x{:}a\,.\,x : a \to a}\,(\to \mathrm{i}-1)}{\lambda a\,.\,\lambda x{:}a\,.\,x : (\forall a)(a \to a)}\,(\forall \mathrm{i})}{(\lambda a\,.\,\lambda x{:}a\,.\,x)\beta : \beta \to \beta}\,(\forall \mathrm{e})$$

Note that the term in the conclusion reduces to $\lambda x{:}\beta\,.\,x$.

For the further theory of TAP, including the normalization theorem, see Fortune et al. [FLO83] and Mitchell [Mit86]. For a proof of the Church-Rosser theorem for the reduction defined in Definition 10, see van Daalen [Daa80], § II.6.

2.4 The power of second order quantification

It might appear that the next order of business is to add the type forming operators $\times$ and $+$ and to arrange to add the new atomic type $\mathbf{N}$. However, these additions turn out to be unnecessary; for all of these can be defined, as can their associated functions.

Definition 2.13 (Cartesian product type) Let α and β be any two type schemes in TAP, and let a be a type variable which does not occur free in α or β. Then the *product type scheme* $\alpha \times \beta$ and its associated pairing and projection operators are defined as follows:
(a) $\alpha \times \beta \equiv (\forall a)((\alpha \to (\beta \to a)) \to a)$;
(b) $\mathsf{D}_{\alpha,\beta} \equiv \lambda x{:}\alpha \,.\, \lambda y{:}\beta \,.\, \lambda a \,.\, \lambda z{:}\alpha \to (\beta \to a) \,.\, zxy$;
(c) $\mathsf{fst}_{\alpha,\beta} \equiv \lambda x{:}\alpha \times \beta \,.\, x\alpha(\lambda u{:}\alpha \,.\, \lambda v{:}\beta \,.\, u)$; and
(d) $\mathsf{snd}_{\alpha,\beta} \equiv \lambda x{:}\alpha \times \beta \,.\, x\beta(\lambda u{:}\alpha \,.\, \lambda v{:}\beta \,.\, v)$.

It is not at all difficult to prove that from these definitions we have

$$\mathsf{D}_{\alpha,\beta} : \alpha \to (\beta \to \alpha \times \beta),$$

$$\mathsf{fst}_{\alpha,\beta} : \alpha \times \beta \to \alpha,$$

and

$$\mathsf{snd}_{\alpha,\beta} : \alpha \times \beta \to \beta.$$

Furthermore, we can easily see that

$$\mathsf{fst}_{\alpha,\beta}(\mathsf{D}_{\alpha,\beta} M N) =_* M$$

and

$$\mathsf{snd}_{\alpha,\beta}(\mathsf{D}_{\alpha,\beta} M N) =_* N.$$

Definition 2.14 (Disjoint union type) Let α and a be any two type schemes in TAP, and let a be a type variable which does not occur free in α or β. Then the disjoint union type scheme $\alpha + \beta$ and its associated injection and case operators are defined as follows:
(a) $\alpha + \beta \equiv (\forall a)((\alpha \to a) \to ((\beta \to a) \to a))$;
(b) $\mathsf{inl}_{\alpha,\beta} \equiv \lambda x{:}\alpha \,.\, \lambda a \,.\, \lambda f{:}\alpha \to a \,.\, \lambda g{:}\beta \to a \,.\, fx$;
(c) $\mathsf{inr}_{\alpha,\beta} \equiv \lambda y{:}\beta \,.\, \lambda a \,.\, \lambda f{:}\alpha \to a \,.\, \lambda g{:}\beta \to a \,.\, gy$;
(d) $\mathsf{case}_{\alpha,\beta} \equiv \lambda x{:}\alpha + \beta \,.\, \lambda a \,.\, \lambda f{:}\alpha \to a \,.\, \lambda g{:}\beta \to a \,.\, zafg$.

It is easy to show that these definitions imply

$$\mathsf{inl}_{\alpha,\beta} : \alpha \to \alpha + \beta,$$

$$\mathsf{inr}_{\alpha,\beta} : \beta \to \alpha + \beta,$$

and

$$\mathsf{case}_{\alpha,\beta} : \alpha + \beta \to (\forall a)((\alpha \to a) \to ((\beta \to a) \to a)).$$

Furthermore, it is easy to show that if γ is any type scheme and if M, N, F, and G are any terms assigned types α, β, $\alpha \to \gamma$, and $\beta \to \gamma$ respectively, then

$$\mathsf{case}_{\alpha,\beta}(\mathsf{inl}_{\alpha,\beta}M)\gamma FG =_* FM$$

and

$$\mathsf{case}_{\alpha,\beta}(\mathsf{inr}_{\alpha,\beta}N)\gamma FG =_* GN.$$

It turns out that we can also define the type void:

Definition 2.15 (Void type) $\mathsf{void} \equiv (\forall a)a$.

Then if M : void, and if α is any type, then $M\alpha : \alpha$. It follows that if M is any closed term such that M : void, and if θ is any type constant, then $M\theta$ is a closed term assigned type θ . This together with the normalization theorem prove the following result:

Theorem 2.4 *There is no closed term M such that*

$$\vdash_{\mathrm{TAP}} M : \mathsf{void}.$$

We can also define the natural number type N:

Definition 2.16 (Natural number type) (a) $\mathsf{N} \equiv (\forall a)((a \to a) \to (a \to a))$;
(b) $\mathbf{0} \equiv \lambda a\,.\,\lambda x{:}a \to a\,.\,\lambda y{:}a\,.\,y$;
(c) $\sigma \equiv \lambda u{:}\mathsf{N}\,.\,\lambda a\,.\,\lambda x{:}a \to a\,.\;\lambda y{:}a\,.\,x(uaxy)$;
(d) $\pi \equiv \lambda u{:}\mathsf{N}\,.\,\mathsf{snd}_{\mathsf{N},\mathsf{N}}(u(\mathsf{N} \times \mathsf{N})\,Q(\mathsf{D}_{\mathsf{N},\mathsf{N}}\mathbf{00}))$,
where $Q \equiv \lambda v : \mathsf{N} \times \mathsf{N}\,.\;\mathsf{D}_{\mathsf{N},\mathsf{N}}(\sigma(\mathsf{fst}_{\mathsf{N},\mathsf{N}}v))(\mathsf{fst}_{\mathsf{N},\mathsf{N}}v)$; and
(e) $\mathsf{R} \equiv \lambda a\,.\,\lambda x{:}a\,.\,\lambda y{:}\mathsf{N} \to a \to a\,.\,\lambda z{:}\mathsf{N}\,.\,z(\mathsf{N} \to a)P(\lambda w : \mathsf{N}\,.\,x)z$,
where $P \equiv \lambda v : \mathsf{N} \to a\,.\,\lambda w : \mathsf{N}\,.\,y(\pi w)(v(\pi w))$. The term n, which represents the natural number n, is defined to be

$$\sigma(\sigma(...(\sigma\mathbf{0})...)),$$

where there are n occurrences of σ.

It is not hard to show that

$$\mathbf{0} : \mathbf{N},$$

$$\sigma : \mathbf{N} \to \mathbf{N},$$

$$\pi : \mathbf{N} \to \mathbf{N},$$

and

$$\mathsf{R} : (\forall a)(a \to (\mathbf{N} \to a \to a) \to \mathbf{N} \to a).$$

It is also easy to show that

$$\mathbf{n} =_* \lambda a \,.\, \lambda x{:}a \to a \,.\, \lambda y{:}a \,.\, x(x(\ldots(xy)\ldots)),$$

where there are n occurrences of x after the last abstraction,

$$\pi\mathbf{0} =_* \mathbf{0},$$

$$\pi(\sigma\mathbf{n}) =_* \mathbf{n},$$

and also, for any type scheme α and any terms M and N of types α and $\mathbf{N} \to \alpha \to \alpha$ respectively,

$$\mathsf{R}\alpha MN\mathbf{0} =_* M,$$

and

$$\mathsf{R}\alpha MN(\sigma\mathbf{n}) =_* N\mathbf{n}(\mathsf{R}\alpha MN\mathbf{n}).$$

Finally, we can define an existential quantifier over all types to go along with our universal quantifier.

Definition 2.17 (Existential quantifier over all types) Let β be any type scheme, and let a be a type variable, which may occur free in β. Then the *existential quantifier over all types* and its associated operators are defined as follows:

(a) $(\exists a)\beta \equiv (\forall b)((\forall a)(\beta \to b) \to b)$,
(b) $\mathsf{single}_\beta \equiv \lambda c \,.\, \lambda x{:}[c/a]\beta \,.\, \lambda b \,.\, \lambda z{:}(\forall a)(\beta \to b) \,.\, zcx$,
(c) $\mathsf{project}_\beta \equiv \lambda x{:}(\exists a)\beta \,.\, \lambda b \,.\, \lambda z{:}(\forall a)(\beta \to b) \,.\, xbz$.

It is easy to show that

$$\mathsf{single}_\beta : (\forall c)([c/a]\beta \to (\exists a)a)$$

and

$$\mathsf{project}_\beta : (\exists a)\beta \to (\forall b)((\forall a)(\beta \to b) \to b).$$

It is also easy to show that if α and γ are type schemes in which a does not occur free and if M and F are terms assigned types $[\alpha/a]\beta$ and $(\forall a)(\beta \to \gamma)$ respectively, then

$$\mathsf{project}_\beta(\mathsf{single}_\beta \alpha M)\gamma F =_* F\alpha M.$$

Thus, we can think of single_β as a kind of singleton, or one-tuple, in which the object has type $[\alpha/a]\beta$, and $\mathsf{project}_\beta$ is as close as we can come to a projection function. Note that the type for single_β tells us that if M is a term of type $[\alpha/a]\beta$, then $\mathsf{single}_\beta \alpha M$ is in type $(\exists a)\beta$, and the type for $\mathsf{project}_\beta$ tells us that if M is a term of type $(\exists a)\beta$, γ is any type scheme in which a does not occur free, and F is any term of type $(\forall a)(\beta \to \gamma)$, then $\mathsf{project}_\beta M \gamma F$ is in type γ; this gives us one of the important properties of existence in logic, as we shall see in Section 3.5.

It might appear that we can obtain a true projection function by forming $\mathsf{project}_\beta N \gamma F$ where $F\alpha M =_* M$. But this fails to work, for in this case F must be the term

$$\lambda a \,.\, \lambda x{:}[\alpha/a]\beta \,.\, x,$$

which has type $(\forall a)([\alpha/a]\beta \to [\alpha/a]\beta)$, which means that α must be a and γ must be $[\alpha/a]\beta$, which is just β itself; thus, a occurs free in both α and γ, which violates the conditions for the type of $\mathsf{project}_\beta$ given above.

Note Most of the terms defined in this subsection which have type schemes as parameters can be defined as terms representing functions applied to these type schemes. For example, if we define

$$\mathsf{D} \equiv \lambda a \,.\, \lambda b \,.\, \mathsf{D}_{a,b},$$

then for any type schemes α and β,

$$\mathsf{D}\alpha\beta =_* \mathsf{D}_{\alpha,\beta}.$$

This idea also works for fst, snd, inl, inr, case and R. It fails to work for single_β and $\mathsf{project}_\beta$ because of the type variable which occurs free in β (in the interesting cases) and which is bound in the definitions. Furthermore, since we do not have in TAP any machinery for representing functions whose values are types, we cannot do a similar thing for $\alpha \times \beta$ or $\alpha + \beta$.

2.5 Generalized type assignment

Although the two term-forming operators $\rightarrow$ and $\forall$ may appear to be entirely distinct, they can be made special instances of a more general type forming operator. This more general operator is central to the theory of constructions.

This more general operator is obtained by extending the meaning of "type" in TA by defining $(\forall x : \alpha)\beta$ to be a type whenever α and β are types and x does not occur free in α. Here, x may occur free in β. Thus, the notion of type used here is much more general than the notion of type in TA. But let us ignore this for the moment and look at the elimination and introduction rules for these types, which are as follows:

$$(\forall\alpha\ \text{e})\quad \frac{M : (\forall x : \alpha)\beta \qquad N : \alpha}{MN : [N/x]\beta,}$$

$$(\forall\alpha\ \text{i})\quad \frac{\begin{array}{c}[x : \alpha]\\ M : \beta\end{array}}{\lambda x{:}\alpha\ .\ M : (\forall x : \alpha)\beta.}$$

Condition: x does not occur free in α or in any undischarged assumption.

If x does not occur free in β, then $(\forall x : \alpha)\beta$ behaves just like $\alpha \rightarrow \beta$, and the above rules become $(\rightarrow$ e) and $(\rightarrow$ i). Hence, if $(\forall x : \alpha)\beta$ is a type whenever α and β are types, then $\alpha \rightarrow \beta$ can be defined to be $(\forall x : \alpha)\beta$ for a variable x which does not occur free in either α or β.

Systems like this are called systems of generalized type assignment, and are covered in Hindley & Seldin [HS86] Chapter 16 and in the references given there. Note that the notation is different there, since what we are denoting by $(\forall x : \alpha)\beta$ is there denoted by $G\alpha(\lambda x.\beta)$, and what is there denoted by $G\alpha\beta$ is here denoted by $(\forall x : \alpha)(\beta x)$.

As we noted above, the definition of type needed for this sort of system is much more complicated than that used in TA. In TA it is sufficient to define types, and except for type variables there are no variables which occur in types. But here, in order to have a system which is really more interesting than TA, it is necessary to have types in which term variables occur. This means, in effect, that we need not only types, but also functions whose values are types. Hence, any formalism for generalized type assignment must include terms representing such functions.

Systems of generalized type assignment can be classified by the kinds of functions they have whose values are types, and in particular by what kinds of domains

such functions can have. The simplest assumption to make about such functions is that the domains are all universal; i.e., if α is any type function of n arguments and M is any term whatsoever, then αM is a type function of $n-1$ arguments (where, of course, $n \geq 1$). A system of this sort is called basic generalized type assignment, and we shall look at such systems in Section 2.7. The only alternative is to allow functions whose values are types over restricted domains. One possibility, for example, is to allow functions whose values are types when the arguments are natural numbers, but not necessarily otherwise. Including functions of this kind complicates the definition of the systems: either the definition of type and type function must list each restricted domain used, or else the machinery of type assignment itself must be used to define the functions involved. We shall see more about this in Section 2.8.

2.6 The need for conversion rules

Before we proceed, we need to consider the question of conversion. In TA, we have the subject-reduction theorem (Theorem 2.1), which says that type assignment is invariant of reduction. As we shall see below, a similar result holds for generalized type assignment. For this reason, we have not paid attention to conversions among *terms* to which types are assigned. Furthermore, in TA, the structure of the types is so simple that the question of conversions between types just does not come up. But in generalized type assignment, the structure of types is more complicated, and so interesting conversions arise.

The best example of this can be seen in terms of the system TAGU of Section 2.8 below (Definition 2.24). Suppose one of the types is U of that system, and suppose we internalize the definition of $\rightarrow$ (which we discussed in Section 5) as follows (using Curry's notation):

$$\mathsf{F} \equiv \lambda u{:}\mathsf{U} \,.\, \lambda v{:}\mathsf{U} \,.\, (\forall x : u)v.$$

It is not hard to show that F has type $(\forall u : \mathsf{U})(\forall v : \mathsf{U})\mathsf{U}$. Now suppose we have, for $\alpha : \mathsf{U}$ and $\beta : \mathsf{U}$,

$$M : \mathsf{F}\alpha\beta$$

and

$$N : \alpha$$

We would like to be able to conclude

$$MN : \beta.$$

However, to do this with our rules requires

$$M : (\forall x : \alpha)\beta,$$

whereas all we have is

$$M : (\lambda u{:}\mathsf{U} \,.\, \lambda v{:}\mathsf{U} \,.\, (\forall x : u)v)\alpha\beta.$$

It is true that this latter type converts to $(\forall x : \alpha)\beta$, but with the rules we have so far this is no help.

To solve this problem, we introduce the following rule:

$$(\mathrm{Eq}'') \qquad \frac{M : \alpha \qquad \alpha =_* \beta}{M : \beta}$$

(On the reason for the name of this rule, see Hindley & Seldin [HS86] Section 14E.)

This rule is often written as follows:

$$\frac{M : \alpha}{M : \beta}\ (\mathrm{Eq}'')$$

It is easy to reconstruct the right premise.

It might appear that the introduction of this rule significantly complicates the nature of deductions and raises problems with the subject-construction theorem. But in fact it is possible to limit the places in which this rule is used:

Theorem 2.5 *In a system of generalized type assignment in which the rules are* $(\forall\alpha\ \mathrm{e})$, $(\forall\alpha\ \mathrm{i})$, $(\equiv'_\alpha)$ *and* (Eq''), *(and in which there may be axioms), any deduction can be transformed into another deduction with the same undischarged assumption and conclusion in which each inference by rule* (Eq'')*occurs either just above the major (left) premise for an inference by rule* $(\forall\alpha\ \mathrm{e})$ *or else just above the conclusion.*

Proof This follows from the fact that the following transformations can be carried out systematically throughout any deduction:

I.

$$\dfrac{\dfrac{\begin{array}{c}1\\ [x : \alpha]\\ \mathcal{D}\\ M : \beta\end{array}}{M : \gamma}\ (\mathrm{Eq}'')}{\lambda x{:}\alpha\ .\ M : (\forall x : \alpha)\gamma}\ (\forall\alpha\ \mathrm{i}-1)$$

to

$$\dfrac{\dfrac{\begin{array}{c}1\\ [x : \alpha]\\ \mathcal{D}\\ M : \beta\end{array}}{\lambda x{:}\alpha\ .\ M : (\forall x : \alpha)\beta}\ (\forall\alpha\ \mathrm{i}-1)}{\lambda x{:}\alpha\ .\ M : (\forall x : \alpha)\gamma}\ (\mathrm{Eq}'')$$

II.

$$\dfrac{\dfrac{\mathcal{D}_1}{M : (\forall x : \beta)\gamma} \qquad \dfrac{\dfrac{\mathcal{D}_2}{N : \alpha}}{N : \beta}\,(\mathrm{Eq}'')}{MN : [N/x]\gamma}\,(\forall\alpha\ \mathrm{e})$$

to

$$\dfrac{\dfrac{\dfrac{\mathcal{D}_1}{M : (\forall x : \beta)\gamma}}{M : (\forall x : \alpha)\gamma}\,(\mathrm{Eq}'') \qquad \dfrac{\mathcal{D}_2}{N : \alpha}}{MN : [N/x]\gamma}\,(\forall\alpha\ \mathrm{e})$$

III.

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{M : \alpha}}{M : \beta}\,(\mathrm{Eq}'')}{N : \beta}\,(\equiv'_\alpha)$$

to

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{M : \alpha}}{N : \alpha}\,(\equiv'_\alpha)}{N : \beta}\,(\mathrm{Eq}'')$$

■

2.7 Basic generalized type assignment

As we noted in Section 2.5, the simplest form of generalized type assignment assumes that any term can be any argument of any type-valued function. The system based on this assumption is called *basic generalized type assignment*, abbreviated TAG.

The first step in defining this system is to define the terms and the types. In this case, the types will all be terms, so we begin with the terms. Because type functions will take any terms as arguments, it turns out to be convenient not to carry along in the notation the type of each bound variable.

Definition 2.18 (TAG terms) The terms of TAG are defined from countably many term variables $x_1, x_2, \ldots, x_n, \ldots$, and some term constants, including a finite or infinite sequence of constants $\theta_1, \theta_2, \ldots$, as follows:

(a) every term variable and term constant is a term;

(b) if M and N are terms, then so is (MN); and

(c) if x is a term variable and A and M are terms, then $(\lambda x.M)$ and $(\forall x : A)M$ are terms.

With each constant θ_i is associated a non-negative integer $\mathrm{dg}(\theta_i)$ called its *degree*. The constants θ_i are called *type constants*.

Reduction for TAG terms will be defined as in Definition 1.6; The only possible contractions in a term of the form $(\forall x : A)M$ will be those which take place entirely inside A and M.

Now we can define the types and type functions. Each type function will have a *rank* (the number of occurrences of $\forall$) and a *degree*[2]. The types will be the type functions of degree 0.

Definition 2.19 (Atomic type function) A term α is said to be an *atomic type function of degree* n if and only if

$$\alpha \equiv \theta M_1 M_2 \ldots M_k,$$

where θ is a type constant of degree $k + n$ and $M_1, M_2, \ldots, M_k$ are any terms.

Definition 2.20 (Proper TAG type functions) The term α is a *proper TAG type function of rank* m *and degree* n if and only if one of the following conditions

[2]The number of arguments needed to produce a type. The degree of a type constant is a special case of the degree of an atomic type function, which, in turn, is a special case of the degree of a type function.

is met:

(a) α is an atomic type function of degree n and $m = 0$;

(b) $\alpha \equiv \lambda x.\beta$, where β is a proper TAG type function of rank m and degree $n - 1$ (and where, of course, $n > 0$);

(c) $\alpha \equiv (\forall x : \beta)\gamma$, where β and γ are proper TAG type functions of degree 0, $n = 0$, and $m = 1 + \text{rank}(\beta) + \text{rank}(\gamma)$.

Definition 2.21 (TAG type functions) The term α is a TAG *type function of rank m and degree n* if and only if there is a proper TAG type function β of rank m and degree n such that $\alpha\beta$. A TAG *type* is a TAG type function of degree 0.

Theorem 2.6 *The degree and rank of a TAG type function are unique. Furthermore, TAG type functions have the following properties:*

T1. *If α is a TAG type function of rank m and degree n and if β is any term such that $\alpha =_* \beta$, then β is a TAG type function of rank m and degree n;*

T2. *If α is a TAG type function of rank m and degree n, then $\lambda x.\alpha$ is a TAG type function of rank m and degree $n + 1$, and conversely;*

T3. *If α is a TAG type function of rank m and degree $n + 1$ and if M is any term, then αM is a TAG type function of rank m and degree n; and*

T4. *$(\forall x : \alpha)\beta$ is a TAG type function of rank m and degree 0 if and only if α and β are TAG type functions of ranks j and k respectively and degree 0 and $m = 1+j+k$.*

Proof See Hindley & Seldin [HS86] Theorem 16.27 and Remark 16.28. ■

Definition 2.22 (The type assignment system TAG) The system TAG is a natural deduction system. Its formulas have the form

$$M : \alpha,$$

where M is a term and α is a TAG type. TAG has no axioms. Its rules are $(\forall\alpha\ \text{e})$, $(\forall\alpha\ \text{i})$, (Eq″) and $(\equiv'_\alpha)$.

Remark It might seem unnecessary to postulate rule (Eq″) here, since the argument of Section 2.6 does not apply to this system. But it is traditional to postulate it, especially since in the earliest versions $(\forall x : \alpha)\beta$ was only an abbreviation for $\text{G}\alpha(\lambda x.\beta)$, and rule $(\forall\alpha\ \text{e})$ had to be obtained from the following rule:

$$\frac{M : \text{G}\alpha\beta \qquad N : \alpha}{MN : \beta N.}$$

To obtain our rule ($\forall\alpha$ e) from this rule requires rule (Eq″); indeed, to use the elimination rule given here in a nontrivial way requires rule (Eq″). See Hindley & Seldin [HS86] Section 16D2.

The theory of TAG is similar to the theory of TA (Section 2.1). There are some complications, but for the case we are considering here they are not serious. For example, rules (Eq″) and ($\equiv'_\alpha$) complicate the subject-construction property, but a version of the property holds (see Hindley & Seldin [HS86] Remark 16.37). The replacement lemma (Lemma 2.1) needs some modification, but a version of it can be proved that will work with the subject-reduction theorem (Theorem 2.1), which holds for β-reduction. (Hindley & Seldin [HS86] Lemma 16.39 and Theorem 16.41). The normalization theorem for deductions (Theorem 2.2) also holds (Hindley & Seldin [HS86] Theorem 16.45).

In fact, TAG is not much stronger than TA. It can be shown that if a term is assigned a type by TAG, then it is assigned a type by TA, although TAG may assign more general types. (See Hindley & Seldin [HS86] Theorem 16.61.) And if all of the type constants have degree 0, then TAG is equivalent to TA (Hindley & Seldin [HS86] Corollary 16.61.1). These facts may appear to show that TAG is too weak to be interesting. Perhaps it is better to take them as showing that TAG is a kind of conservative extension of TA, and thus that the basic formalism on which TAG is based is sound. This can give us some confidence in extending TAG, as we now proceed to do in the next section.

2.8 Extended generalized type assignment

As we noted at the end of Section 2.1, there are two ways to generalize TAG: one is to modify the definition of type to allow certain special types (such as the type N of natural numbers) to serve as restricted domains for type functions, and the other is to use the machinery of type assignment itself to define the types. Since the second approach is obviously more general, we shall adopt it here.

Thus, we now suppose that that there is a type of types, or a "universal" type, which for now we shall call U. All the types in which we are interested will be in U. The system we shall define here will be called "TAGU". The reasons we had for not supplying the type of a bound variable no longer apply, so we shall return to the more familiar notation.

Definition 2.23 (TAGU terms) The terms of TAGU are defined from countably many *term variables* x_1, x_2, ..., x_n, ..., and some *term constants*, which include U, as follows:

(a) every term variable and term constant is a term;

(b) if M and N are terms, then so is (MN); and

(c) if x is a term variable and A and M are terms, then $(\lambda x : A.M)$ and $(\forall x : A)M$ are terms.

Reduction for TAGU terms will be defined using the β^1-redexes of Definition 2.11. The only possible contractions in a term of the form $(\forall x : A)M$ are those which take place entirely inside A and M.

Definition 2.24 (The type assignment system TAGU) The system TAGU is a natural deduction system. Its formulas have the form

$$M : A$$

where M and A are terms. It has no axioms. Its rules are (Eq''), ($\equiv'_\alpha$), and the following:

Rules of type formation:

(∀ Formation)
$$\frac{A : \mathsf{U} \qquad \begin{matrix}[x : A] \\ B : \mathsf{U}\end{matrix}}{(\forall x : A)B : \mathsf{U}}$$
Condition: x does not occur free in A or in any undischarged assumption.

(Eq′U)
$$\frac{A : \mathsf{U} \qquad A =_* B}{B : \mathsf{U}}$$

Rules of type assignment:

(∀ e)
$$\frac{M : (\forall x : A)B \qquad N : A}{MN : [N/x]B}$$

(∀Ui)
$$\frac{\begin{matrix}[x : A] \\ M : B\end{matrix} \qquad A : \mathsf{U}}{\lambda x{:}A \,.\, M : (\forall x : A)B}$$
Condition: x does not occur free in A or in any undischarged assumption.

Rule (Eq′U) is a natural rule to go with rule (Eq″). We can extend the proof of Theorem 2.5 to virtually eliminate it from any deduction.

Theorem 2.7 *Every deduction in TAGU can be transformed into a deduction with the same undischarged assumptions and conclusion in which each inference by either of rules* (Eq″) *and* (Eq′U) *occurs just above the major (left) premise for an inference by rule* (Eq′U) (*in which case it is an inference by rule* (Eq″)) *or just above the minor (right) premise for an inference by rule* (∀Ui) (*in which case it is an inference by rule* (Eq′U)) *or just above the conclusion.*[3]

Proof Note that each rule which discharges an assumption of the form $x : A$ has a premise of the form $A : \mathsf{U}$ which does not depend on the discharged assumption. Let us call the deduction of this latter premise the *independent subdeduction* of the

[3]Note that it is possible to have an inference by rule (Eq′U) followed immediately by an inference by rule (Eq″), the conclusion of which is the conclusion of the deduction. In this case, the inference by rule (Eq′U) will be regarded as occurring just above the conclusion.

rule and the deduction of the other premise the *dependent subdeduction.* The proof is obtained by transformations which move an inference by one of the equality rules from an independent subdeduction of a rule to the dependent subdeduction of the same rule or else to below the conclusion, from a dependent subdeduction to below the conclusion, from just above a minor premise of ($\forall$ e) to just above the major premise, or from just above an inference by ($\equiv'_\alpha$) to below the conclusion. If an inference by rule (Eq″) occurs just above an inference by rule (Eq′U), then the transformations moving the latter inference are applied before an attempt is made to move the former (since clearly, an inference by rule (Eq″) occurring just above an inference by rule (Eq′U) cannot be moved below it without invalidating it). The last two kinds of transformations are II and III of Theorem 2.5; in addition, we now need the following transformations:

IV.

$$\dfrac{\dfrac{\mathcal{D}_1}{\dfrac{C : \mathsf{U}}{A : \mathsf{U}}\,(\mathrm{Eq'U})} \qquad \begin{matrix}1\\ [x : A]\\ \mathcal{D}_2(x)\\ B : \mathsf{U}\end{matrix}}{\begin{matrix}(\forall x : A)B : \mathsf{U}\\ \mathcal{D}_3\end{matrix}}\,(\forall\ \text{Formation} - 1)$$

to

$$\dfrac{\mathcal{D}_1 \quad \dfrac{\dfrac{\dfrac{1}{[x:C]}}{x:A}\,(\mathrm{Eq}'')}{\mathcal{D}_2(x)}}{\dfrac{C:\mathsf{U} \qquad B:\mathsf{U}}{(\forall x:C)B:\mathsf{U}}\,(\forall\ \mathrm{Formation}-1)}\,(\mathrm{Eq}'\mathsf{U})}{(\forall x:A)B:\mathsf{U}}$$

$$\mathcal{D}_3$$

V.

$$\dfrac{\mathcal{D}_1 \quad \dfrac{\dfrac{1}{[x:A]}}{\mathcal{D}_2(x)}}{\dfrac{A:\mathsf{U} \qquad C:\mathsf{U}}{B:\mathsf{U}}\,(\mathrm{Eq}'\mathsf{U})}\,(\forall\ \mathrm{Formation}-1)}{(\forall x:A)B:\mathsf{U}}$$

$$\mathcal{D}_3$$

to

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \dfrac{1}{[x:A]}}{\mathcal{D}_2(x)}}{\dfrac{A:\mathsf{U} \qquad C:\mathsf{U}}{(\forall x:A)C:\mathsf{U}}\,(\forall\ \mathrm{Formation}-1)}\,(\mathrm{Eq}'\mathsf{U})}{(\forall x:A)B:\mathsf{U}}$$

$$\mathcal{D}_3$$

VI.

$$\dfrac{\dfrac{\begin{array}{c}1\\ [x:A]\\ \mathcal{D}_1(x)\\ M:C\end{array}}{M:B}\,(\mathrm{Eq}'')\qquad \begin{array}{c}\mathcal{D}_2\\ A:\mathsf{U}\end{array}}{\begin{array}{c}\lambda x{:}A\,.\,M:(\forall x:A)B\\ \mathcal{D}_3\end{array}}\,(\forall \mathsf{U}\ \mathrm{i}-1)$$

to

$$\dfrac{\dfrac{\begin{array}{c}1\\ [x:A]\\ \mathcal{D}_1(x)\\ M:C\end{array}\qquad \begin{array}{c}\mathcal{D}_2\\ A:\mathsf{U}\end{array}}{\lambda x{:}A\,.\,M:(\forall x:A)C}\,(\forall \mathsf{U}\ \mathrm{i}-1)}{\begin{array}{c}\lambda x{:}A\,.\,M:(\forall x:A)B\\ \mathcal{D}_3\end{array}}\,(\mathrm{Eq}'')$$

■

If we try to remove an inference by rule (Eq′U) just above the right premise of (∀Ui) the best we can do is the transformation which takes

$$\dfrac{\begin{array}{cc}1 & \\ [x:A] & \mathcal{D}_2\\ \mathcal{D}_1 & C:\mathsf{U}\\ M:B & \dfrac{}{A:\mathsf{U}}\,(\mathrm{Eq}'\mathsf{U})\end{array}}{\begin{array}{c}\lambda x{:}A\,.\,M:(\forall x:A)B\\ \mathcal{D}_3\end{array}}\,(\forall \mathsf{U}\ \mathrm{i}-1)$$

to

$$\dfrac{\dfrac{\dfrac{1}{[x:C]}}{x:A}\ (\mathrm{Eq}'')\qquad \mathcal{D}_1 \quad \mathcal{D}_2}{\dfrac{\dfrac{M:B \qquad C:\mathsf{U}}{\lambda x{:}C\,.\,M:(\forall x:C)B}\ (\forall \mathsf{U}\ \mathrm{i}-1)}{\lambda x{:}C\,.\,M:(\forall x:A)B}\ (\mathrm{Eq}'')}\qquad \mathcal{D}_3$$

Note that this transformation changes the type of the bound variable in the term to the left of the colon, and therefore cannot be used with this theorem.

This system is a part of the type theory of Martin-Löf, and is, in fact, one of the most important parts; see the references listed under his name. At the same time, the system has some weaknesses. For example, it is weaker than TAP: the condition $A : \mathsf{U}$ in rule (∀Ui)prevents inferences corresponding to those by rule (∀i)in TAP because $\mathsf{U} : \mathsf{U}$ does not hold.[4] There are several ways one might extend this system. One might follow Martin-Löf himself by introducing more universes. Thus, the type U would become U_0, and a new sequence of types U_1, U_2, ..., U_n, ...(finitely or infinitely many) would be introduced with axioms such as $\mathsf{U}_n : \mathsf{U}_{n+1}$ and rules such as the following:

$$\frac{A:\mathsf{U}_n}{A:\mathsf{U}_{n+1}}$$

Then in rules (∀ Formation) and (∀Ui), U may be replaced by any U_n. But this system is still weaker than TAP.

Another way to extend TAGU is to add two more rules: the formation rule

$$\frac{\begin{array}{c}[x:\mathsf{U}]\\ A:\mathsf{U}\end{array}}{(\forall x:\mathsf{U})A:\mathsf{U}}$$

Condition: x does not occur free in any undischarged assumption.

[4] In fact, adding $\mathsf{U} : \mathsf{U}$ to TAGU makes the system inconsistent; see [Coq86a].

and the type assignment rule

$$\frac{\begin{array}{c}[x : \mathsf{U}] \\ M : A\end{array}}{\lambda x{:}\mathsf{U} \,.\, M : (\forall x : \mathsf{U})A.}$$

Condition: x does not occur free in any undischarged assumption.

This system is called TAGL in Hindley & Seldin [HS86] §16E, since there U is called L. Furthermore, TAP can be interpreted in this system. Nevertheless, the system is still not as strong as one might want, since one might wonder why not allow $x : \mathsf{U} \to \mathsf{U}$ as the discharged assumption.

In Chapter 4, we shall consider the theory of constructions, introduced by Coquand [Coq85]. This turns out to be the best available system of this kind. (See Chapter 4 for further references.)

Chapter 3

CONSTRUCTIVE LOGIC

A reader who has read this far is now in a position to understand the basic rules and the metatheory of the theory of constructions. However, there is an important aspect of the theory of constructions that we have not discussed; it has to do not with the underlying rules but rather with its intended interpretation. This interpretation is an important part of the motivation Coquand had in creating the system. Some readers might find it useful to consider this interpretation before proceeding to the theory of constructions itself. For this reason, the theory of constructions will be postponed to Chapter 4, and in this chapter we will consider that interpretation.

The interpretation is what is usually known as the *Curry-Howard isomorphism*, or *formulas-as-types* idea. The essence of it is that in systems of type assignment, types can be thought of as formulas and terms as proofs or deductions. We will consider this here for constructive logic, and it is with this that we will begin (in the latter part of this introduction). In Section 3.1, we take up a simple fragment of the propositional calculus for constructive logic in which the only logical connective is $\supset$ (if-then). In Section 3.2, we explain the essentials of the formulas-as-types idea. For some readers, this may be enough, and these readers are invited to proceed to Chapter 4 after completing Section 3.2.

For readers who want more, we consider in Sections 3.3-3.4 the extension of these ideas to propositional calculus with the additional connectives $\wedge$ (and), $\vee$ (or), and $\neg$ (not). Again, many readers may wish to proceed to Chapter 4 after completing Section 3.4.

But for those who want still more, we consider in Sections 3.5-3.6 the extension of these ideas to predicate logic, both first order logic (Section 3.5) and higher order logic[1] (Section 3.6). The systems TAJ and TAT presented in these sections

[1] I.e., simple type theory.

will seem strange to some people, and they are not strictly necessary for using the theory of constructions, but they do give some useful information about much of its motivation and intended interpretation.

Let us now turn our attention to constructive logic. Most people who have heard of constructive logic understand that it has something to do with existence proofs. But in fact, the difference between classical and constructive logic involves more than that. In classical logic we are only interested in whether or not a proposition is true. In constructive logic we are interested in whether or not a proposition has a proof, and we do not want to assert its provability without having access to a proof.

This difference can be illustrated with formulas involving implication. A formula $A \supset B$ is classically false when A is true and B is false; it is true for all other combinations of truth values for A and B. Note that its truth value depends only on the truth values of A and B; how these truth values are established is classically irrelevant.

In constructive logic, implication is not truth functional; the truth of $A \supset B$ depends on much more than the truth values of A and B. In fact, instead of specifying when $A \supset B$ is true, we need to specify what it means to have a proof of $A \supset B$. The standard constructive specification is as follows: a *proof* of $A \supset B$ is a function [program] which, given any proof of A as an argument [input], produces a proof of B as a value [output].

Truth in classical logic (at least propositional logic) can be defined by means of truth tables. In constructive logic, however, we really need to introduce a kind of calculus of proofs.

3.1 The ⊃-calculus

One way of defining a system of formal logic that seems especially suited to constructive logic is to use a *natural deduction system* of the kind introduced by Jaśkowski [Jas34] and Gentzen [Gen34] and studied extensively by Prawitz [Pra65] . We have seen the method of writing rules used by Gentzen and Prawitz in Section 2.1, but we have not really discussed natural deduction systems as such. In a natural deduction system, each logical constant is characterized by two rules, one for introducing it and one for eliminating it. In the case of implication, these two rules are as follows:

$$(\supset \text{e}) \quad \frac{A \supset B \qquad A}{B} \qquad\qquad (\supset \text{i}) \quad \frac{\begin{array}{c}[A]\\ B\end{array}}{A \supset B}$$

Rule (⊃ e) is also known as *modus ponens*, and rule (⊃ i) is sometimes called the *deduction theorem.*

A formal calculus of propositional logic for the constructive theory of ⊃ can be defined as follows:

Definition 3.1 (⊃-formulas) Assume that there are (finitely or countably many) *atomic formulas* E_1, E_2, ..., E_n, ... Then ⊃*-formulas*, or *formulas* are defined as follows:

(a) Every atomic formula is a formula;

(b) If A and B are formulas, then so is $(A \supset B)$. Unnecessary parentheses will be omitted. Furthermore,

$$A_1 \supset A_2 \ldots A_n \supset B$$

will be regarded as an abbreviation for

$$A_1 \supset (A_2 \supset (\ldots(A_n \supset B)\ldots)).$$

Definition 3.2 (The formal calculus NA(⊃)) The formal calculus NA(⊃)[2]is a natural deduction system. Its formulas are ⊃-formulas. It has no axioms; its rules are (⊃ e) and (⊃ i) given above.

Here are some examples of deductions in NA(⊃), given in table form:

[2]The name NA(⊃) means the implication fragment of NA. Here the "N" stands for "natural deduction", while "A" stands for "absolute", a term used by Curry [Cur63] to stand for constructive logic without negation. (Curry, who was using "N" for negation, called the system TA, but here this would be confused with "type assignment". The letter "N" was used in this way by Gentzen [Gen34].)

Example 3.1 $\vdash_{NA(\supset)} A \supset A$

Proof.

1.	A	Hyp	1
2.	$A \supset A$	1 ($\supset$ e)	

Example 3.2 $\vdash_{NA(\supset)} A \supset B \supset A$

Proof.

1.	A	Hyp	1
2.	$B \supset A$	1 ($\supset$ i)	1
3.	$A \supset B \supset A$	2 ($\supset$ i)	

Example 3.3 $\vdash_{NA(\supset)} (A \supset B \supset C) \supset (A \supset B) \supset A \supset C$

Proof.

1.	$A \supset B \supset C$	Hyp	1
2.	$A \supset B$	Hyp	2
3.	A	Hyp	3
4.	$B \supset C$	1, 3 ($\supset$ e)	1, 3
5.	B	2, 3 ($\supset$ e)	2, 3
6.	C	4, 5 ($\supset$ e)	1, 2, 3
7.	$A \supset C$	6 ($\supset$ i)	1,2
8.	$(A \supset B) \supset A \supset C$	7 ($\supset$ i)	1
9.	$(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$	8 ($\supset$ i)	

Example 3.4 $A \supset B, B \supset C \vdash_{NA(\supset)} A \supset C$

Proof.

1.	$A \supset B$	Hyp	1
2.	$B \supset C$	Hyp	2
3.	A	Hyp	3
4.	B	1, 3 ($\supset$ e)	1, 3
5.	C	2, 4 ($\supset$ e)	1, 2, 3
6.	$A \supset C$	5 ($\supset$ i)	1, 2

In tree form, the examples are as follows:

Example 3.1′

$$\dfrac{\dfrac{\overset{1}{[A]}}{A \supset A}\ (\supset \text{i} - 1)}{}$$

Example 3.2′

$$\dfrac{\dfrac{\dfrac{\overset{1}{[A]}}{B \supset A}\ (\supset \text{i} - \text{v})}{A \supset B \supset A}\ (\supset \text{i} - 1)}{}$$

Example 3.3′

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{1}{[A \supset B \supset C]} \quad \overset{3}{[A]}}{B \supset C}\ (\supset \text{e}) \quad \dfrac{\overset{2}{[A \supset B]} \quad \overset{3}{[A]}}{B}\ (\supset \text{e})}{C}\ (\supset \text{e})}{A \supset C}\ (\supset \text{i} - 3)}{(A \supset B) \supset A \supset C}\ (\supset \text{i} - 2)}{(A \supset B \supset C) \supset (A \supset B) \supset A \supset C}\ (\supset \text{i} - 1)}{}$$

Example 3.4′

$$\dfrac{\dfrac{\dfrac{\text{Hyp} \quad \overset{\text{Hyp}}{A \supset B} \quad \overset{1}{[A]}}{B \supset C \qquad\qquad B}\ (\supset \text{e})}{C}\ (\supset \text{e})}{A \supset C}\ (\supset \text{i} - 1)}{}$$

3.2 Formulas-as-types

If Definition 3.1 is compared with the remarks immediately before Definition 1.3 (in Section 1.2), it will be observed that the $\supset$-formulas are isomorphic to the type symbols used in defining the basic typed λ-terms; each atomic formula E_i corresponds to an atomic type θ_i, and if A and B correspond to α and β respectively, then $A \supset B$ corresponds to $\alpha \rightarrow \beta$. If Definition 3.2 is compared with Definition 2.3, it should be clear that deductions in NA($\supset$) are isomorphic to deductions in TA. Now by the subject-construction theorem, the terms in deductions in TA are isomorphic to the deductions. Hence, we can think of TA as a calculus of deductions of NA($\supset$), where the types represent the formulas and the terms represent the deductions. If we make use of Definition 2.3, we can use basic typed λ -terms to represent deductions in NA($\supset$).

This correspondence between typed λ-calculus and propositional logic was first noticed by Curry in [CF58] Section 9E, and was later extended independently by a number of people, including W. A. Howard [How80]. (For more references, see Hindley & Seldin [HS86] Discussion 14.46.) The correspondence is usually called *formulas-as-types isomorphism* or the *Curry-Howard isomorphism.*

As we noted after Definition 2.3, a β-reduction step for deductions in TA is similar to the $\supset$-reduction step of Prawitz [Pra65]. In fact, under the formulas-as-types isomorphism, the two types of reduction steps correspond exactly, the proof of Theorem 2.2 (i.e., the proof of Theorem 1.2) together with the isomorphism proves Prawitz's result for NA($\supset$), namely that every deduction can be reduced to a normal form. Here, a normal form means that nowhere in the deduction is the conclusion of an inference by ($\supset$ i) the major (left) premise for an inference by ($\supset$ e).

This isomorphism can also be used to show that certain formulas are not provable in NA($\supset$). Let us consider as an example the formula known as *Peirce's law*:

$$((A \supset B) \supset A) \supset A.$$

It is not hard to see that this formula is classically true, for it is only necessary to consider what assignment of truth values could make it false. This would require an assignment that makes A false and $(A \supset B) \supset A$ true. Now if A is false and $(A \supset B) \supset A$ is true, then $A \supset B$ must also be false, but this is impossible if A is false. Thus, Perice's law is always assigned the value true by a truth table. Nevertheless, it is not constructively valid.

Theorem 3.1 *The formula scheme* $((A \supset B) \supset A) \supset A$ *is not provable in NA*($\supset$).

Proof If this formula were provable, it would be the conclusion of a normal deduction in which every assumption is discharged. By the formulas-as-types isomor-

phism, it would follow that for any two types α and β, there is a closed term M in normal form such that

$$\vdash_{\mathrm{TA}} M : ((\alpha \to \beta) \to \alpha) \to \alpha.$$

It follows that $M : ((\alpha \to \beta) \to \alpha) \to \alpha$ is the conclusion of a deduction $\mathcal{D}$ in normal form. By the subject-construction theorem, M must have the form $\lambda x.N$ for some term N for which $\mathrm{FV}(N) \subseteq \{x\}$, and $\mathcal{D}$ must have the form

$$\dfrac{\begin{array}{c} 1 \\ [x : (\alpha \to \beta) \to \alpha] \\ \mathcal{D}_1 \\ N : \alpha \end{array}}{\lambda x.N : ((\alpha \to \beta) \to \alpha) \to \alpha.} \ (\to \mathrm{i} - 1)$$

Since it is sufficient to prove that there exist types α and β for which this is impossible, there is no loss of generality in assuming that α is atomic, and thus that there is no inference by $(\to \mathrm{i})$ in the left branch of $\mathcal{D}_1$. Since the only undischarged assumption in $\mathcal{D}_1$ is $x : (\alpha \to \beta) \to \alpha$, it follows that this assumption occurs at the top of the left branch of $\mathcal{D}_1$. Hence, $\mathcal{D}_1$ has the following form, where N is xP:

$$\dfrac{\begin{array}{cc} & x : (\alpha \to \beta) \to \alpha \\ & \mathcal{D}_2 \\ x : (\alpha \to \beta) \to \alpha & P : \alpha \to \beta \end{array}}{xP : \alpha} \ (\to \mathrm{e})$$

Note that $\mathrm{FV}(P) \subseteq \{x\}$. Now consider the structure of $\mathcal{D}_2$: if the left branch had no inference by $(\to \mathrm{i})$, then the left branch would begin with the assumption $x : (\alpha \to \beta) \to \alpha$ and would end with $P : \alpha \to \beta$, which is impossible since α is assumed to be atomic. It follows that $\mathcal{D}_2$ has the following form, where P is $\lambda y.Q$:

$$\dfrac{\begin{array}{c} 2 \\ x : (\alpha \to \beta) \to \alpha, \quad [y : \alpha] \\ \mathcal{D}_3 \\ Q : \beta \end{array}}{\lambda y.Q : \alpha \to \beta} \ (\to \mathrm{i} - 2)$$

Hence, $\mathcal{D}_3$ is a normal deduction of

$$x : (\alpha \to \beta) \to \alpha,\ y : \alpha \ \vdash_{\mathrm{TA}} \ Q : \beta,$$

where $\mathrm{FV}(Q) \subseteq \{x, y\}$. Since we can assume without loss of generality that β as well as α is atomic, this is clearly impossible. ■

Corollary 3.1.1 *If A and B are atomic formulas, then*

$$\forall_{\mathrm{NA}(\supset)}\ ((A \supset B) \supset A) \supset A.$$

3.3 Adding $\wedge$,$\vee$, and $\bot$ (for $\neg$)

Let us now turn to the full propositional calculus. In addition to $\supset$ (implication), we need $\wedge$ (and), $\vee$ (or), and $\neg$ (not). In constructive logic, $\neg$ is usually defined in terms of $\bot$ (absurdity), and we shall follow this practice here.

Definition 3.3 (Propositional formulas) Assume that, as in Definition 3.1, we have finitely or countably many given atomic formulas $E_1, \ldots, E_n, \ldots$. *Propositional formulas* are then defined as follows:

(a) a given atomic formula E_i is an (atomic) formula;

(b) $\bot$ is an (atomic) formula; and

(c) if A and B are formulas, then so are $(A \supset B)$, $(A \wedge B)$, and $(A \vee B)$.

Notation Unnecessary parentheses will be omitted. The infixes $\wedge$ and $\vee$ will have smaller scope than $\supset$. The abbreviation

$$\neg A$$

will be used for

$$A \supset \bot .$$

The elimination and introduction rules postulated for $\wedge$ and $\vee$ are as follows:

($\wedge$e) $$\frac{A \wedge B}{A,} \qquad \frac{A \wedge B}{B}$$

($\wedge$i) $$\frac{A \quad B}{A \wedge B}$$

($\vee$e) $$\frac{A \vee B \qquad \begin{matrix}[A] & [B] \\ C & C\end{matrix}}{C}$$

($\vee$i) $$\frac{A}{A \vee B,} \qquad \frac{B}{A \vee B}$$

Of these rules, ($\vee$e) will probably look least familiar. It is easy to understand if we think of proof by cases: if case A or case B holds, and if C can be proved in each case, then C must be provable.

The elimination and introduction rules for negation, which are derived from those for implication, are as follows:

$$(\neg e)\ \frac{\neg A \qquad A}{\bot} \qquad\qquad (\neg i)\ \frac{\begin{array}{c}[A]\\ \bot\end{array}}{\neg A}$$

There is one additional rule used with negation: it is as follows:

$$(\bot\ j)\ \frac{\bot}{A}$$

It expresses the fact that anything follows from a contradiction, a fact accepted by most constructivists. (For those constructivists who do not accept this principle, there is the *minimal calculus*, which is the system NJ without this rule. We will not bother with the minimal calculus here.)

This leads us to the following definition:

Definition 3.4 (The formal calculus NJ) The formal calculus NJ is a natural deduction system. Its formulas are the propositional formulas of Definition 3.3. It has no axioms. Its rules are ($\supset$ e), ($\supset$ i), ($\wedge$e), ($\wedge$i), ($\vee$e), ($\vee$i), and ($\bot$ j).

Remark Many people may be surprised that rule ($\neg$i) is constructively valid, since it is often said that constructivists object to proof by contradiction. In fact, the form of proof by contradiction to which constructivists object is not ($\neg$i), but rather the following rule:

$$(\bot\ d)\ \frac{\begin{array}{c}[\neg A]\\ \bot\end{array}}{A}$$

This rule is not valid in NJ; in fact, if it is added to NJ, the result is classical logic.

It turns out that it is possible to modify Definition 3.4 somewhat:

Lemma 3.1 *If rule* ($\perp$ j) *is postulated in the form*

$$\frac{\perp}{E,}$$

where E is one of the given atomic formulas, then the rule holds in its full generality as a derived rule.

Proof Since the case of the rule in which A is $\perp$ is trivial, it is sufficient to prove the rule for compound formulas A on the assumption that it holds for shorter formulas. The three cases (note that $\neg$ is taken care of by the case for $\supset$) are taken care of by the following three deductions:

$$\dfrac{\dfrac{\perp}{B}\ (\perp\,\mathrm{j})}{A \supset B}\ (\supset\,\mathrm{i}-\mathrm{v})$$

$$\dfrac{\dfrac{\perp}{A}\ (\perp\,\mathrm{j}) \qquad \dfrac{\perp}{B}\ (\perp\,\mathrm{j})}{A \wedge B}\ (\wedge\mathrm{i})$$

$$\dfrac{\dfrac{\perp}{A}\ (\perp\,\mathrm{j})}{A \vee B}\ (\vee\mathrm{i})$$

■

3.4 Extension of formulas-as-types

In order to extend the formulas-as-types isomorphism of Section 2 to NJ, it is most natural to compare $\wedge$, $\vee$, and $\perp$ to $\times$, $+$, and void. This leads us to consider the system extended TA of the remark at the end of Section 2.1. But this system does not correspond exactly to NJ. Instead it corresponds to a system obtained from NJ by replacing the rules ($\wedge$e), ($\wedge$i), ($\vee$e), and ($\vee$i) by the following axiom schemes:

(1) $A \supset B \supset A \wedge B$;

(2) $A \wedge B \supset A$;

(3) $A \wedge B \supset B$;

(4) $A \supset A \vee B$;

(5) $B \supset A \vee B$;

and

(6) $A \vee B \supset (A \supset C) \supset (B \supset C) \supset C$.

It should be clear that, in the presence of the rules ($\supset$ e) and ($\supset$ i), these six axiom schemes are equivalent to the indicated rules.

Note that by Lemma 3.1, rule ($\perp$ j) is equivalent to the scheme

(7) $\perp \supset E$,

where E is an atomic formula distinct from $\perp$. This scheme would appear not to correspond to any term in extended TA, since such a term would have to be assigned the type void $\rightarrow \theta$ for an atomic type θ. If there is some object M in the type θ, then we can apply ($\rightarrow$ i) with vacuous discharge of the assumption x : void to obtain the conclusion $\lambda x.M$: void $\rightarrow \theta$. But we cannot guarantee that there is an object M to which θ is assigned for each atomic type θ; indeed, if there were such a term for each atomic type, this would correspond to the provability of each atomic formula. So instead, we will add to extended TA a constant $\perp_\theta$ for each atomic type θ distinct from void, and we will assume the axiom

$$(\perp \text{ j}_\theta)\ \perp_\theta : \text{void} \rightarrow \theta.$$

Since these constants $\perp_\theta$ do not occur at the beginning of any redexes, they do not affect the normalization result. Hence, these axioms cannot be used to produce closed terms in any of the θ. Furthermore, by the proof of Lemma 3.1, it should be clear that for each type α there is a closed term $\perp_\theta$ of type void $\rightarrow \alpha$.

It is not difficult to show that Theorem 3.1 and Corollary 3.1.1 apply to NJ. The normalization theorem for extended TA plus the constants $\perp_\theta$ and axioms ($\perp$ j$_\theta$) can be used to prove that NJ is, indeed, different from classical logic in one of its most important aspects.

Theorem 3.2 *For at least one formula A*

$$\not\vdash_{NJ} A \vee \neg A.$$

Proof Let A be an atomic formula. Let $\mathcal{D}$ be a proof (i.e., a deduction with no undischarged assumptions) whose conclusion is $A \vee \neg A$. An instance of axiom scheme (6) is

$$A \vee \neg A \supset (A \supset A) \supset (\neg A \supset A) \supset A.$$

Using this, $\mathcal{D}$, Example 3.1, and two inferences by ($\supset$ e), we get a proof of

$$(\neg A \supset A) \supset A,$$

which is, when abbreviations are removed,

$$((A \supset \bot) \supset A) \supset A.$$

Since both A and $\bot$ are atomic formulas, this is unprovable by Corollary 3.1.1.[3] ■

[3]The reduction and normalization procedure used here for NJ, which is based on extended TA plus ($\bot$ j_θ), is not the usual normalization procedure for NJ in proof theory. For the usual procedure, see Prawitz [Pra65] Chapter IV.

3.5 First order quantifiers

It is standard in logic to proceed from propositional logic to first order logic. In first order logic, universal and existential quantifiers are present, and are assumed to operate over one fundamental domain of individuals; it is not possible to quantify over sets of individuals or functions whose arguments and values are individuals.

To take an example from elementary arithmetic, suppose that the fundamental domain is the set of natural numbers, and suppose that our language has terms representing the natural numbers and also addition and multiplication (which, for now, will be denoted by their usual notation in algebra). Suppose also that formulas include equations between expressions denoting numbers. Then a formula stating that x is an even number is

$$(\exists y)(x = \mathbf{2}y),$$

where **2** is the term representing the number 2. A formula stating that $x < y$ is

$$(\exists u)(\neg u = \mathbf{0} \wedge y = x + u),$$

where **0** represents the number 0. (Recall that in the set of natural numbers, there are no negative numbers, so that if a number is different from 0 it is positive.) A formula which says that x divides evenly into y is

$$(\exists u)(\neg u = \mathbf{0} \wedge y = xu).$$

Finally, a formula which says that 0 is an identity for addition is

$$(\forall x)(x = x + \mathbf{0}).$$

In giving these examples, I assumed that there is a term representing each natural number. In fact, such terms are easy to construct: begin with an *individual constant* **0** and a *function symbol* σ with one argument. Then the term **n** representing the natural number n is

$$\sigma(\sigma(\ldots(\sigma\mathbf{0})\ldots)),$$

where there are n occurrences of σ.

If we analyze the structure of the formulas in these examples, we see that we have an *individual constant* **0**, *individual variables* x, y, u, ..., *function symbols* σ of one argument and $+$ and $\cdot$ (multiplication) of two arguments, a *predicate symbol* $=$ of two arguments, the logical connectives of propositional logic, and the universal and existential quantifiers. This leads us to the following formal definition:

Definition 3.5 (First order term and formula) Assume that we have countably many *individual variables* x, y, z, x_1, etc., finitely or countably many *individual constants* e_1, e_2,..., finitely or countably many *function symbols* $\omega_1, \omega_2, \ldots$, and finitely or countably many *predicate symbols* φ_1, φ_2, ..., where each function symbol and predicate symbol has associated with it a natural number called its *degree*, which represents its number of arguments. Then *terms* are defined as follows:

(a) individual constants and individual variables are terms; and

(b) if ω is a function symbol of degree m, and if $t_1, \ldots, t_m$ are terms, then $\omega(t_1, \ldots, t_m)$ is a term.

First order formulas are now defined as follows:

(c) if φ is a predicate symbol of degree m and if $t_1, \ldots, t_m$ are terms, then $\varphi(t_1, \ldots, t_m)$ is an atomic formula;

(d) $\perp$ is an atomic formula;

(e) if A and B are formulas, then so are $(A \wedge B)$, $(A \vee B)$, and $(A \supset B)$; and

(f) if A is a formula and x an individual variable, then $(\forall x)A$ and $(\exists x)A$ are formulas. Parentheses will be omitted as usual. An occurrence of an individual variable is said to be *bound* if it is within the scope of a universal or existential quantifier; otherwise it is *free*.

Notes (1) Both function symbols and predicate symbols may have degree 0. A function symbol of degree 0 is just an individual constant; individual constants are listed separately because it is customary to do so. A predicate symbol of degree 0 is an atomic formula. One example of such an atomic formula is $\perp$.

(2) Here $\perp$ is, in effect, taken to be a predicate symbol of degree 0. But this is not necessary in all first order systems. For example, in first order arithmetic, $\perp$ is often defined to be the atomic formula $\mathbf{0} = \sigma\mathbf{0}$, which is $\mathbf{0} = \mathbf{1}$. What is important is that $\perp$ be an *atomic* formula.

Definition 3.6 (The formal calculus NJ*) The formal calculus NJ* is a natural deduction system. Its formulas are the first order formulas of Definition 3.5. It has no axioms. Its rules are the rules of NJ and, in addition, the following:

$$(\forall e) \qquad \frac{(\forall a)A(x)}{A(t)} \qquad \textit{Condition: } t \text{ is a term.}$$

(∀i)	$\dfrac{A(x)}{(\forall a)A(x)}$	*Condition:* x does not occur free in any undischarged assumption.
(∃e)	$\dfrac{(\exists x)A(x) \qquad [A(y)]}{C}$	*Condition:* y does not occur free in C or in any undischarged assumption.
(∃i)	$\dfrac{A(t)}{(\exists x)A(x)}$	*Condition:* t is a term.

The condition on the variable x in rule (∀i) guarantees that no assumption is made about x above the inference. Rule (∃e) formalizes the argument: there is an x such that $A(x)$; let y be a thing such that $A(y)$; conclusion C (where y does not occur free in C). See the discussion after Definition 2.17. The condition on y is obviously necessary for this rule. Variables such as x in (∀i) and y in (∃e) are called *eigenvariables or characteristic variables.*

At first glance it might appear that the natural way to extend the formulas-as-types isomorphism to NJ* is to use the system TAP. But this will not work. For in TAP, only types (corresponding to formulas) can be substituted for the (type) variables, whereas in NJ* we must be able to substitute terms for the quantified variables. Instead, we will need to take a type to represent the fundamental domain of quantification, and introduce quantification over that type. We will also need to modify the definition of type to correspond to Definition 3.5.

Thus, suppose one of the atomic types is J, the type of individuals. For each atomic constant e, we will want to assume

$$e : \mathsf{J}.$$

For each function symbol ω of degree m, we will want to assume

$$\omega : \mathsf{J} \to \mathsf{J} \to \ldots \to \mathsf{J},$$

where there are $m + 1$ occurrences of J. Then it will follow for each closed term t that

$$t : \mathsf{J}.$$

Furthermore, if t is a term with free variables $x_1, \ldots, x_n$, then it will follow that

$$x_1 : \mathsf{J}, \ldots, x_n : \mathsf{J} \vdash t : \mathsf{J}.$$

Next, we need to generalize the definition of atomic type: for each predicate symbol φ of degree m, and for any terms $t_1, \ldots, t_m$, we need that $\varphi(t_1, \ldots, t_m)$ is a type. We also assume void is an atomic type, and form as usual types $\alpha \times \beta$, $\alpha + \beta$, and $\alpha \to \beta$. Also, we need that if x is a variable and α is a type, then $(\forall x : \mathsf{J})\alpha$ and $(\exists x : \mathsf{J})\alpha$ are types.

It remains to specify the terms in $(\forall x : \mathsf{J})\alpha$ and $(\exists x : \mathsf{J})\alpha$. For the type $(\forall x : \mathsf{J})\alpha$, we want a function which, when applied to any object t of type J, produces a value in $[t/x]\alpha$. Note that as in TAG the type of this function depends on its argument and not just on the type of its argument. For $(\exists x : \mathsf{J})\alpha$, we want to have pairs $\langle t, M \rangle$ such that t has type J and M has type $[t/x]\alpha$. These are just the kind of pairs we were unable to represent in the type structures of Section 1.1. We shall have more to say about this later.

The above conventions, although stated as in previous definitions, can also be obtained by using the machinery of TA or TAG. What is necessary is some type to which the above types belong, such as the type U of Section 2.8. Since the above types represent propositions, this new type will be called Prop. We have the following formal definition:

Definition 3.7 (TAJ types) The *types* of the system TAJ are defined as follows:

(a) J and Prop are (atomic) types; and

(b) if α and β are types, then so is $(\alpha \to \beta)$. The special types J^n and Prop^n for $n \geq 0$ are defined as follows (by induction on n):

$$\begin{array}{ll} \mathsf{J}^0 \equiv \mathsf{J}, & \mathsf{J}^{n+1} \equiv \mathsf{J} \to \mathsf{J}^n; \\ \mathsf{Prop}^0 \equiv \mathsf{Prop}, & \mathsf{Prop}^{n+1} \equiv \mathsf{J} \to \mathsf{Prop}^n. \end{array}$$

Definition 3.8 (TAJ terms) The *terms* of TAJ are defined from countably many *term variables* $x_1, x_2, \ldots, x_n, \ldots$, and the *term constants* $e_1, e_2, \ldots$, $\omega_1, \omega_2, \ldots$, $\varphi_1, \varphi_2, \ldots$, void, D, D_J, fst, snd, inl, inr, case, proj_J, and, $\perp$, as follows:

(a) every term variable and term constant is a term;

(b) if M, N, A, and B are terms, so are (MN),$(A \times B)$, $(A + B)$, and $(A \to B)$; and

(c) if x is a term variable and A and M are terms, then $(\lambda x{:}A \,.\, M)$, $(\lambda x{:}\mathsf{J} \,.\, M)$, $(\forall x : \mathsf{J})A$, and $(\exists x : \mathsf{J})A$ are terms. With each constant ω_i and φ_i is associated a natural number $\mathrm{dg}(\omega_i)$ or $\mathrm{dg}(\varphi_i)$, called the *degree* of the constant in question.

Definition 3.9 (Reduction for TAJ terms) *Reduction* for TAJ terms is defined by the following table of redexes and contracta:

	Redex	Contractum
(β)	$(\lambda x : A.M)N$	$[N/x]M$
(fst)	$\mathsf{fst}AB(\mathsf{D}ABMN)$	M
(snd)	$\mathsf{snd}AB(\mathsf{D}ABMN)$	N
(case_1)	$\mathsf{case}AB(\mathsf{inl}ABM)CFG$	FM
(case_2)	$\mathsf{case}AB(\mathsf{inr}ABM)CFG$	GM
(proj)	$\mathsf{proj}_J ACZ(\mathsf{D}_J AMN)$	ZMN

Definition 3.10 (The type assignment system TAJ) The system TAJ is a natural deduction system. Its formulas are all expressions of the form

$$M : A,$$

where M is a term and A is either a term or a type. The *axioms* are as follows:

(e_i) $\quad e_i : \mathsf{J}$,

(ω_i) $\quad \omega_i : \mathsf{J}^m, \quad m = \mathrm{dg}(\omega_i)$,

(φ_i) $\quad \varphi_i : \mathsf{Prop}^m, \quad m = \mathrm{dg}(\varphi_i)$,

for each i and

(void) $\quad \mathsf{void} : \mathsf{Prop}$

The *rules* of TAJ come in two groups:

Rules of type formation:

$$(\times \text{ Formation}) \quad \frac{A : \mathsf{Prop} \qquad B : \mathsf{Prop}}{A \times B : \mathsf{Prop}}$$

$$(+ \text{ Formation}) \quad \frac{A : \mathsf{Prop} \qquad B : \mathsf{Prop}}{A + B : \mathsf{Prop}}$$

$$(\rightarrow \text{ Formation}) \quad \frac{A : \mathsf{Prop} \qquad B : \mathsf{Prop}}{A \rightarrow B : \mathsf{Prop}}$$

($\forall$JFormation)
$$\frac{[x : \mathsf{J}] \quad A : \mathsf{Prop}}{(\forall x : \mathsf{J})A : \mathsf{Prop}}$$
Condition: x does not occur free in any undischarged assumption.

($\exists$JFormation)
$$\frac{[x : \mathsf{J}] \quad A : \mathsf{Prop}}{(\exists x : \mathsf{J})A : \mathsf{Prop}}$$
Condition: x does not occur free in any undischarged assumption.

Rules of type assignment:

($\times$ e)$_1$
$$\frac{M : A \times B \quad A : \mathsf{Prop} \quad B : \mathsf{Prop}}{\mathsf{fst}ABM : A}$$

($\times$ e)$_2$
$$\frac{M : A \times B \quad A : \mathsf{Prop} \quad B : \mathsf{Prop}}{\mathsf{snd}ABM : B}$$

($\times$ i)
$$\frac{M : A \quad N : B \quad A : \mathsf{Prop} \quad B : \mathsf{Prop}}{\mathsf{D}ABMN : A \times B}$$

(+ e)
$$\frac{M : A + B \quad \lambda x{:}A \,.\, N : C \quad [y : B] \quad P : C \quad A : \mathsf{Prop} \quad B : \mathsf{Prop} \quad C : \mathsf{Prop}}{\mathsf{case}ABMC(\lambda x{:}A \,.\, N)(\lambda y{:}B \,.\, P) : C}$$
Condition: x and y do not occur free in M, A, B, C, or in any undischarged assumption; x does not occur free in P, and y does not occur free in N.

$(+\,\mathrm{i})_1$ $$\frac{M : A \quad A : \mathsf{Prop} \quad B : \mathsf{Prop}}{\mathsf{inl}ABM : A + B}$$

$(+\,\mathrm{i})_2$ $$\frac{N : B \quad A : \mathsf{Prop} \quad B : \mathsf{Prop}}{\mathsf{inr}ABN : A + B}$$

$(\rightarrow \mathrm{e})$ $$\frac{M : A \rightarrow B \qquad N : A}{MN : B}$$

Condition: A and B are both terms or both types.

$(\rightarrow \mathrm{i})_1$ $$\frac{\begin{array}{l}[x : A] \\ M : B \qquad A : \mathsf{Prop}\end{array}}{\lambda x{:}A \,.\, M : A \rightarrow B}$$

Condition: x does not occur free in A, B, or in any undischarged assumption, and A is a term.

$(\rightarrow \mathrm{i})_2$ $$\frac{\begin{array}{c}[x : A] \\ M : B\end{array}}{\lambda x{:}A \,.\, M : A \rightarrow B}$$

Condition: x does not occur free in A, B, or in any undischarged assumption, and A and B are types.

$(\perp\,\mathrm{j}\varphi_i)$ Foreach$i$, $$\frac{N_1 : \mathsf{J} \quad N_2 : \mathsf{J} \quad \ldots \quad N_m : \mathsf{J}}{\perp\ \varphi_i N_1 N_2 \ldots N_m : \mathsf{void} \rightarrow \varphi_i N_1 N_2 \ldots N_m}$$

Condition: $m = \mathrm{dg}(\varphi_i)$.

$(\forall \mathrm{Je})$ $$\frac{M : (\forall x : \mathsf{J})A \qquad N : \mathsf{J}}{MN : [N/x]A}$$

$(\forall \mathrm{Ji})$ $$\frac{\begin{array}{c}[x : \mathsf{J}] \\ M : A\end{array}}{\lambda x{:}\mathsf{J} \,.\, M : (\forall x : \mathsf{J})A}$$

Condition: x does not occur free in any undischarged assumption.

(∃Je)

$$\frac{M : (\exists x : \mathsf{J})A \qquad N : C \qquad \begin{matrix}[x : \mathsf{J}][y : A] \\ A : \mathsf{Prop}\end{matrix} \qquad \begin{matrix}[x : \mathsf{J}] \\ C : \mathsf{Prop}\end{matrix}}{\mathsf{proj}_{\mathsf{J}}(\lambda x{:}\mathsf{J} \,.\, A)C(\lambda x{:}\mathsf{J} \,.\, \lambda y{:}A \,.\, N)M : C}$$

Condition: x and y do not occur free in C, M, or in any undischarged assumptions, and y does not occur free in A.

(∃Ji)

$$\frac{M : J \quad N : [M/x]A \qquad \begin{matrix}[x : J] \\ A : \mathsf{Prop}\end{matrix}}{\mathsf{D}_{\mathsf{J}}(\lambda x{:}\mathsf{J} \,.\, A)MN : (\exists x : \mathsf{J})A}$$

Condition: x does not occur free in M or N or in any undischarged assumption.

$(\equiv'_\alpha)$

$$\frac{M : A}{N : A}$$

Condition: N is obtained from M by changes of bound variables.

$(\equiv'''_\alpha)$

$$\frac{M : A}{M : B}$$

Condition: B is obtained from A by changes of bound variables.

Notes (1) As we have seen, we have in TAJ functions the type of whose values depend on the arguments as well as the types of the arguments, and we also have pairs in which the type of the second element depends on the first element as well as on its type. This means that the type structures of Section 1.1 are not models of TAJ (just as they are not models of TAP). It is possible to construct a kind of semantics for TAJ as follows: J is interpreted as the set of all closed terms of NJ*; Prop is interpreted as the set of closed formulas of NJ*; the function types built up from J and Prop using $\rightarrow$ are interpreted using terms and formulas in which free variables occur; and terms assigned as types terms in Prop are interpreted as deductions or, if they are closed, as proofs. Any other model for TAJ is likely to be

too complicated to provide most people with any insight.

(2) The presence of $\lambda x{:}\mathsf{J}\,.\,A$ in the conclusion of rules ($\exists$Je) and ($\exists$Ji) may seem a bit strange. It is there merely to supply A as an argument, and therefore it might seem more appropriate to use simply A. But if we did that, then x would occur free in the conclusion whenever it occurs free in A, which is contrary to the spirit of the system. The only obvious alternative is to postulate $\mathsf{D}_{\mathsf{J},A}$ and $\mathsf{Proj}_{\mathsf{J},A}$ for each formula A, but in this case whether or not a term $\mathsf{D}_{\mathsf{J},A}$ is defined depends on whether or not there is a deduction whose conclusion is $A : \mathsf{Prop}$, and this is also contrary to the spirit of the system. The (proj) contraction of Definition 3.9 shows that it makes no difference whether A or $\lambda x{:}\mathsf{J}\,.\,A$ is used as an argument here, since it disappears in the contraction.

The system TAJ contains the system NJ* in an important sense, for we can easily write $\wedge$, $\vee$, $\supset$, and $\bot$ instead of $\times$, $+$, $\rightarrow$, and void (provided, of course, that the constant $\bot$ of TAJ is renamed). The system NJ* has been given here as a separate system because it is traditional to do so. However, from here on, systems of logic will only be presented with the systems of type assignment with which they are associated by the formulas-as-types isomorphism.

3.6 The full theory of types

An examination of TAJ raises a question: why quantify only over the type J? Why not quantify over other types, such as Prop? In fact, why not quantify over *all* of the TAJ types of Definition 3.7? There is, in fact, no reason at all for not quantifying over all TAJ types, and a logic based on this idea was proposed as long ago as 1940 by Church [Chu40]. A version of this system will now be presented as a system of type assignment.

Clearly the main difference between TAJ and the system that will be defined here is that instead of only $(\forall x : \mathsf{J})$ and $(\exists x : \mathsf{J})$, we will now have $(\forall x : \alpha)$ and $(\exists x : \alpha)$ for every TAJ type α. It should be clear how to obtain the more general quantifier rules required here from those of TAJ.

However, there is another important difference: one of the TAJ types is Prop, and since we can quantify over Prop, we can interpret TAP in this new system. This means that we can use the definitions of Section 2.4 to reduce the number of primitives.

The new system will be called TAT.

The *types* of TAT will be those of TAJ (Definition 3.7).

Definition 3.11 (TAT terms) The *terms* of TAT are defined from countably many *term varialbes* $x_1, x_2, \ldots, x_n, \ldots$, and the *term constants* $e_1, e_2, \ldots, \omega_1, \omega_2, \ldots, \varphi_1, \varphi_2, \ldots$, as follows:

(a) every term variable and term constant is a term;

(b) if M and N, are terms, so are (MN) and $(M \to N)$; and

(c) if x is a term variable, A and M are terms, and α is a type, then $(\lambda x{:}A \,.\, M)$, $(\lambda x{:}\alpha \,.\, M)$, and $(\forall x : A)$ are terms. With each constant ω_i and φ_i is associated a natural number $\mathrm{dg}(\omega_i)$ or $\mathrm{dg}(\varphi_i)$, called the *degree* of the constant in question.

Reduction for TAT terms is defined using the β-redexes of Definition 3.9.

Definition 3.12 (The type assignment system TAT) The system TAT is a natural deduction system. Its formulas are all expressions of the form

$$M : A,$$

where M is a term and A is either a term or a type. The *axioms* are (e_i), (ω_i), and (φ_i) from Definition 3.10 for each i. The *rules of type formation* are ($\to$ Formation)

of Definition 3.10 and

$$(\forall\alpha \text{ Formation}) \quad \frac{[x : \alpha] \quad A : \mathsf{Prop}}{(\forall x : \alpha)A : \mathsf{Prop}}$$

Condition: x does not occur free in any undischarged assumption, and α is a type.

The *rules of type assignment* are ($\rightarrow$ e), ($\rightarrow$ i), ($\equiv'_\alpha$), and ($\equiv'''_\alpha$) of Definition 3.10 and, for each type α,

$$(\forall\alpha\text{e}) \quad \frac{M : (\forall x : \alpha)A \qquad N : \alpha}{MN : [n/x]A}$$

$$(\forall\alpha\text{i}) \quad \frac{[x : \alpha] \quad M : A}{\lambda x{:}\alpha \,.\, M : (\forall x : \alpha)A}$$

Condition: x does not occur free in any undischarged assumption.

Remark As in TAJ, the type structures of Section 1.1 are not models of TAT. There are models of the original (classical) version of Church's type theory formed by interpreting J as any set, Prop as the set of two truth values, true and false, and interpreting compound types $\alpha \rightarrow \beta$ as the set of all functions from the set corresponding to α to the set corresponding to β. But these models are not models of TAT because they do not model the deductions. Furthermore, since TAP can be interpreted in TAT, it follows that TAT has no set theoretic models. It is probably best to adopt the procedure we used for TAJ, and interpret Prop as the set of closed formulas. Because we now have quantifiers over all types, this idea is hard to make precise, and so is unlikely to be accepted as the basis for any kind of *theory* of models. Nevertheless, the idea probably gives most people more insight into TAT than any other notion of semantics.

Now let us show how to use the definitions of Section 2.4 to define the other terms and operators of TAJ. Some changes in the previous definitions will be necessary: wherever we previously had a quantifier $(\forall a)$, we will now need a quantifier $(\forall x : \mathsf{Prop})$, and where we previously used the abstraction λa, we will now need $\lambda u : \mathsf{Prop}$. Furthermore, the existential quantifier will need somewhat different treatment, since

we now expect the elements assigned an existential type will be pairs. In addition, it is now possible to quantify over the parameters that stood for type schemes in TAP and now stand for terms of type Prop. For this reason, it is worth stating these definitions again for this system.

Definition 3.13 (Cartesian product proposition) The *product type operator* and its associated pairing and projection operators are defined as follows:
(a) $\mathsf{X} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, (\forall w : \mathsf{Prop})((u \to v \to w) \to w)$;
(b) $\mathsf{D} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}u \,.\, \lambda y{:}v \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda z{:}u \to v \to w \,.\, zxy$;
(c) $\mathsf{fst} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}\mathsf{X}uv \,.\, xu(\lambda y{:}u \,.\, \lambda z{:}v \,.\, y)$; and
(d) $\mathsf{snd} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}\mathsf{X}uv \,.\, xv(\lambda y{:}u \,.\, \lambda z{:}v \,.\, z)$.
We use $A \times B$ as an abbreviation for $\mathsf{X}AB$.

It is not at all difficult to prove from these definitions that if $A : \mathsf{Prop}$ and $B : \mathsf{Prop}$

$$\mathsf{D}AB : A \to B \to A \times B,$$

$$\mathsf{fst}AB : A \times B \to A,$$

and

$$\mathsf{snd}AB : A \times B \to B.$$

Furthermore, it is easy to see that if $M : A$ and $N : B$, then

$$\mathsf{fst}AB(\mathsf{D}ABMN) =_* M$$

and

$$\mathsf{snd}AB(\mathsf{D}ABMN) =_* N.$$

Definition 3.14 (Disjoint union type) The disjoint union operator and its associated injection and case operators are defined as follows:
(a) $\Theta \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, (\forall w : \mathsf{Prop})((u \to w) \to ((v \to w) \to w))$;
(b) $\mathsf{inl} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}u \,.\, \lambda w{:} : \mathsf{Prop} \,.\, \lambda f{:}u \to w \,.\, \lambda g{:}v \to w \,.\, fx$;
(c) $\mathsf{inr} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda y{:}v \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda f{:}u \to w \,.\, \lambda g{:}v \to w \,.\, gy$; and
(d) $\mathsf{case} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda z{:}\Theta uv \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda f{:}u \to w \,.\, \lambda g{:}v \to w \,.\, zwfg$.
We use $A + B$ as an abbreviation for ΘAB.

It is easy to show that if $A : \mathsf{Prop}$ and $B : \mathsf{Prop}$, then

$$\mathsf{inl}AB : A \to A + B,$$

$$\mathsf{inr}AB : B \to A + B,$$

and

$$\mathsf{case}AB : A + B \to (\forall w : \mathsf{Prop})((A \to w) \to ((B \to w) \to w)).$$

Furthermore, it is easy to show that if C : Prop, $M : A$, $N : B$, $F : A \to C$, and $G : B \to C$, then

$$\mathsf{case}AB(\mathsf{inl}ABM)CFG =_* FM$$

and

$$\mathsf{case}AB(\mathsf{inr}ABN)CFG =_* GN.$$

Definition 3.15 (void type) $\mathsf{void} \equiv (\forall x : \mathsf{Prop})x$.

Definition 3.16 (Existential quantifier) If α is a type, B is a term, and if, for a variable x which does not occur free in α but may occur free in B, we have $x : \alpha \vdash B : \mathsf{Prop}$, then the existential quantifier over α and its associated pairing and projection functions are defined as follows:

(a) $(\exists x : \alpha)B \equiv (\forall w : \mathsf{Prop})((\forall x : \alpha)(B \to w) \to w)$;

(b) $\mathsf{D}_{\alpha,\beta} \equiv \lambda x{:}\alpha\ .\ \lambda y{:}B\ .\ \lambda w{:}\mathsf{Prop}\ .\ \lambda z{:}(\forall x : \alpha)(B \to w)\ .\ zxy$; and

(c) $\mathsf{proj}_{\alpha,\beta} \equiv \lambda w{:}\mathsf{Prop}\ .\ \lambda z{:}(\forall x : \alpha)(B \to w)\ .\ \lambda y{:}(\forall x : \alpha)B\ .\ ywz$.

It not hard to show that rules ($\exists\alpha$Formation), ($\exists\alpha$e) and ($\exists\alpha$i) corresponding to the rules for $\exists$J in Definition 3.10 are satisfied. It is also easy to show that

$$\mathsf{proj}_{\alpha,\beta}CZ(\mathsf{D}_{\alpha,\beta}MN) =_* ZMN.$$

Note that in Definition 3.16, there is no way to avoid the use of the parameters; for types are completely distinct from terms, and there may be a free variable in B which is bound in the definitions.

Remark It is worth comparing $\mathsf{proj}_{\alpha,\beta}$ with $\mathsf{project}_\beta$ of Definition 2.17. For the same reason that the latter could not be made a true projection function, the former cannot be used to define a true right projection for use with rule ($\exists\alpha$e). There is no problem with the left projection: take $C \equiv \alpha$ and take $Z \equiv \lambda x{:}\alpha\ .\ \lambda y{:}B\ .\ x$, and observe that this satisfies the condition on rule ($\exists\alpha$e), which becomes in this case that x and y do not occur free in C or in $\mathsf{D}_{\alpha,\beta}MN$ and y does not occur free in B. On the other hand, for the right projection, we need to take $Z \equiv \lambda x{:}\alpha\ .\ \lambda y{:}B\ .\ B$, and this requires $C \equiv B$, in which x may occur free. Being able to use a right projection with rule ($\exists\alpha$e) would correspond to allowing an inference in NJ* from $(\exists x)A(x)$ to $A(t_A)$ for some term t_A, and making inferences like this work for natural deduction formulations of first order or higher order logic is notoriously difficult.

Chapter 4

THE THEORY OF CONSTRUCTIONS

We have now seen quite a few systems of type assignment to λ-terms. As we said in the introduction, these systems are important for us because they are the basis for the system which really interests us, the *theory of constructions*. This is an extension of TAGU and TAT introduced by Coquand [Coq85] and studied further in [CH86], [CH], [Coq86a], [Coq86b], and [Coq]. We have already seen that TAT is an extension of TAP; the theory of constructions, as an extension of TAT, is also an extension of TAP. It is also an extension of the important part of the *type theory* introduced by Martin-Löf [Mar75], [Mar82], and [Mar84][1]. This chapter will be devoted to the theory of constructions.

The proofs in this chapter will be given in more detail than in previous chapters. This is because the system is new and some of the proofs are difficult. In fact, Martin-Löf [Mar71b] [2] presented a proof of normalization for a system which was later shown not to be normalizable[3]. For this reason, the important proofs in this chapter need to be checked carefully, and so they will be presented in considerable detail.

[1] See also [Bee85] Chapter XI.
[2] An early version of [Mar75].
[3] See [Coq86a].

4.1 The theory of constructions: natural deduction formulation.

The theory of constructions, or TAC, combines the kind of generalized type assignment of systems such as TAG and TAGU with the formulas as types isomorphism used in defining TAT.

As we remarked at the end of Section 2.8, one of the weaknesses we want to eliminate in this system is the fact that in TAGU we cannot quantify over compound types built up from Prop. For this reason, as in TAT, we need a notion of type. But unlike TAP, we cannot define the types as a fixed set of terms. Instead, we need to indicate the types by the rules of the system. Thus, in addition to formulas of the form $M : A$, we need formulas of the form

$$A : \mathsf{Type}$$

The types are then specified by the deductive rules of the system.

Definition 4.1 (TAC terms) The *terms* of TAC are the terms of TAGU (Definition 2.23), where U is denoted by Prop, except that there is a new constant, Type.

The original intention was that Type would not be part of any compound type. However, it has since turned out that it is convenient to have Type occurring as a certain part of certain compound types, as we shall see below.

Definition 4.2 (The type assignment system TAC) The system TAC is a natural deduction system. Its formulas are of the form

$$M : A,$$

where M and A are terms. There is one axiom:

$$\text{(PT)} \qquad \mathsf{Prop} : \mathsf{Type}.$$

The rules are as follows:

Rules of type formation:

(PP*Formation*)
$$\frac{A : \mathsf{Prop} \qquad \begin{matrix}[x : A] \\ B : \mathsf{Prop}\end{matrix}}{(\forall x : A)B : \mathsf{Prop}}$$
Condition: x does not occur free in A or in any undischarged assumption.

(TP*Formation*)
$$\frac{A : \mathsf{Type} \qquad \begin{matrix}[x : A] \\ B : \mathsf{Prop}\end{matrix}}{(\forall x : A)B : \mathsf{Prop}}$$
Condition: x does not occur free in A or in any undischarged assumption.

(PT*Formation*)
$$\frac{A : \mathsf{Prop} \qquad \begin{matrix}[x : A] \\ B : \mathsf{Type}\end{matrix}}{(\forall x : A)B : \mathsf{Type}}$$
Condition: x does not occur free in A or in any undischarged assumption.

(TT*Formation*)
$$\frac{A : \mathsf{Type} \qquad \begin{matrix}[x : A] \\ B : \mathsf{Type}\end{matrix}}{(\forall x : A)B : \mathsf{Type}}$$
Condition: x does not occur free in A or in any undischarged assumption.

(Eq'P)
$$\frac{A : \mathsf{Prop} \qquad A =_* B}{B : \mathsf{Prop}}$$

(Eq'T)
$$\frac{A : \mathsf{Type} \qquad A =_* B}{B : \mathsf{Type}}$$

Rules of type assignment:

(∀e)
$$\frac{M : (\forall x : A)B \qquad N : A}{MN : [N/x]B}$$

(∀Pi) $\dfrac{[x : A] \quad M : B \qquad A : \mathsf{Prop}}{\lambda x{:}A \,.\, M : (\forall x : A)B}$ *Condition:* x does not occur free in A or in any undischarged assumption.

(∀Ti) $\dfrac{[x : A] \quad M : B \qquad A : \mathsf{Type}}{\lambda x{:}A \,.\, M : (\forall x : A)B}$ *Condition:* x does not occur free in A or in any undischarged assumption.

(Eq″) $\dfrac{M : A \qquad A =_* B}{M : B}$

$(\equiv'_\alpha)$ $\dfrac{M : A}{N : A}$ *Condition:* N is obtained from M by changes of bound variables.

(Note that several rules listed earlier are listed here in full: since this system is the main subject of this work, it was felt to be important to make this definition relatively self-contained.)

It is possible to state the rules of this system in a more compact form. To do this, we define the *kinds* to be the two terms Prop and Type. Then if we let κ and κ' be any two kinds, the rules of type formation can be stated as follows:

($\kappa\kappa' Formation$) $\dfrac{A : \kappa \qquad [x : A] \quad B : \kappa'}{(\forall x : A)B : \kappa'}$ *Condition:* x does not occur free in A or in any undischarged assumption.

(Eq′κ) $\dfrac{A : \kappa \qquad A =_* B}{B : \kappa}$

Furthermore, the rules for ($\forall$i) can be combined as follows:

$$(\forall\kappa\text{i})\qquad \frac{\begin{array}{cc}[x:A] & \\ M:B & A:\kappa\end{array}}{\lambda x{:}A\,.\,M:(\forall x:A)B}$$

4.2 The basic metatheory of the theory of constructions

Theorem 2.7 can be extended to TAC:

Theorem 4.1 *Every deduction in TAC can be transformed into a deduction with the same undischarged assumptions and conclusion in which each inference by any of the rules* (Eq″) *and* (Eq′κ) *occurs just above the major (left) premise for an inference by* ($\forall$e) (*in which case it is an inference by rule* (Eq″)) *or just above the minor (right) premise for an inference by* ($\forall\kappa$i) (*in which case it is an inference by rule* (Eq′κ)) *or just above the conclusion.*[4]

Proof Similar to the proof of Theorem 2.7. The definitions of independent subdeduction and dependent subdeduction will be obtained from those of the proof of Theorem 2.7 with U replaced by any kind κ. In addition to transformations II and III from the proof of Theorem 2.5, we need the following transformations (corresponding to transformations IV-VI of the proof of Theorem 2.7):
VII.

$$\dfrac{\dfrac{\mathcal{D}_1}{\dfrac{C:\kappa}{A:\kappa}\,(\mathrm{Eq}'\kappa)} \qquad \begin{matrix}1\\ [x:A]\\ \mathcal{D}_2(x)\\ B:\kappa'\end{matrix}}{(\forall x:A)B:\kappa'}\,(\kappa\kappa'\text{Formation}-1)$$

to

[4] Here, just above the conclusion means what it did in Theorem 2.7, and there may be two such inferences, one by rule (Eq′κ) and the next one by rule (Eq″).

$$\dfrac{\dfrac{\mathcal{D}_1}{C:\kappa}\quad\dfrac{\dfrac{1}{\dfrac{[x:C]}{x:A}\,(\mathrm{Eq}'')}}{\mathcal{D}_2(x)}{B:\kappa'}}{(\forall x:C)B:\kappa'}\,(\kappa\kappa'\text{Formation}-1)}{(\forall x:A)B:\kappa'}\,(\mathrm{Eq}'\kappa')$$

$\mathcal{D}_3$

VIII.

$$\dfrac{\dfrac{\mathcal{D}_1}{A:\kappa}\quad\dfrac{\dfrac{1}{[x:A]}}{\mathcal{D}_2(x)}{C:\kappa'}\,(\mathrm{Eq}'\kappa')}{(\forall x:A)B:\kappa'}\,(\kappa\kappa'\text{Formation}-1)$$

$\mathcal{D}_3$

to

$$\dfrac{\dfrac{\mathcal{D}_1}{A:\kappa}\quad\dfrac{\dfrac{1}{[x:A]}}{\mathcal{D}_2(x)}{C:\kappa'}}{(\forall x:A)C:\kappa'}\,(\kappa\kappa'\text{Formation}-1)}{(\forall x:A)B:\kappa'}\,(\mathrm{Eq}'\kappa')$$

$\mathcal{D}_3$

IX.

$$\dfrac{\dfrac{\begin{array}{c}1\\ [x:A]\\ \mathcal{D}_1(x)\\ M:C\end{array}}{M:B}\,(\mathrm{Eq}'') \qquad \begin{array}{c}\mathcal{D}_2\\ A:\kappa\end{array}}{\begin{array}{c}\lambda x{:}A\,.\,M:(\forall x:A)B\\ \mathcal{D}_3\end{array}}\,(\forall\kappa\mathrm{i}-1)$$

to

$$\dfrac{\dfrac{\begin{array}{c}1\\ [x:A]\\ \mathcal{D}_1(x)\\ M:C\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ A:\kappa\end{array}}{\lambda x{:}A\,.\,M:(\forall x:A)C}\,(\forall\kappa\mathrm{i}-1)}{\begin{array}{c}\lambda x{:}A\,.\,M:(\forall x:A)B\\ \mathcal{D}_3\end{array}}\,(\mathrm{Eq}'')$$

■

From now on, we shall assume without further comment that the transformation given by Theorem 4.1 has been carried out in any deduction. In some cases, when deductions are put together, inferences by equality rules will be indicated at places other than those specified by the theorem; this will mean the deduction obtained from the one shown by carrying out the transformation given by Theorem 4.1.

TAC is clearly an extension of the system TAGU, i.e., of the system TAGL of Hindley & Seldin [HS86] Section 16E. This means that TAP can be interpreted in it.

Theorem 4.2 *TAP can be interpreted in TAC.*

Proof See Hindley & Seldin [HS86] Theorem 16.66. ■

Now let us turn to the general theory of TAC. The first result we have is that Type and Prop control terms which can occur as "types" the way we expect them to. To see this, we need first to consider the conditions under which assumptions may be discharged. For each rule that discharges an assumption of the form $x : A$, there is the independent subdeduction, the conclusion of which is either $A : \mathsf{Prop}$ or $A : \mathsf{Type}$. This fact and the conditions on the occurrences of the variables of discharged assumptions imply that assumptions must be discharged in a certain order. Thus, instead of sets of assumptions, we are really interested in sequences of assumptions. Now suppose that we are given a sequence of assumptions of the form

$$x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n$$

Suppose that the assumption that we wish to discharge is always the last of the sequence. Under what conditions can the last assumption be discharged? And more generally, under what conditions is it always possible to discharge the last assumption of any initial segment of this sequence? It is not difficult to see that the conditions are those of the following definition:

Definition 4.3 ((Well-formed) environments) A (well-formed) environment is a sequence of assumptions

$$x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n \tag{4.1}$$

such that, for $i = 1, 2, ..., n-1$, the following two properties hold:

(a) x_i does not occur free in $A_1, A_2, \ldots, A_i$ (but may occur free in $A_{i+1}, \ldots, A_n$); and

(b) either

$$x_1 : A_1, x_2 : A_2, \ldots, x_i : A_i \vdash_{\mathrm{TAC}} A_{i+1} : \mathsf{Prop}$$

or

$$x_1 : A_1, x_2 : A_2, \ldots, x_i : A_i \vdash_{\mathrm{TAC}} A_{i+1} : \mathsf{Type}.$$

We can now see that the terms which can be proved to be in Type are really quite limited.

Theorem 4.3 *If*

$$\Gamma \vdash_{\mathrm{TAC}} A : \mathsf{Type},$$

for any set of assumptions Γ, *then for some* $n \geq 0$ *and for some terms* $A_1, A_2, \ldots, A_n$, *and for a sequence of pairwise distinct variables* $x_1, x_2, \ldots, x_n$,

$$A =_* (\forall x_1 : A_1)(\forall x_2 : A_2)\ldots(\forall x_n : A_n)\mathsf{Prop}.$$

Proof This follows immediately from the fact that any formula of the form A : Type can occur only as the axiom (P T) or as the conclusion of one of the rules (κT Formation) or (Eq'T). ■

Definition 4.4 (Context) A *context* is a term A satisfying the conclusion of Theorem 4.3. If A is a context, and if the conclusion of Theorem 4.3 is that A is convertible to

$$(\forall x_1 : A_1)(\forall x_2 : A_2)\ldots(\forall x_n : A_n)\mathsf{Prop}, \tag{4.2}$$

then 4.2 is called a *standard form* of A, n is called the *index* of the standard form, and A_1, A_2, ..., A_n are called its *prefix types.*

It is easy to see (by the Church-Rosser theorem) that two standard forms can be standard forms of the same context if and only if they have the same index and corresponding prefix types are convertible. This means that we can speak of the *index of a context*, and if we are willing to consider equivalence classes of convertible terms, we can speak of the *prefix types of a context.* It is also easy to see that any context can be reduced to one of its standard forms.

Contexts have a clear meaning: each context is the type of propositional functions of a certain number of arguments over certain terms as "types". Obviously, contexts are really useful only when the prefix types are either in Prop or in Type. For this reason, we would like to know which contexts can be shown (perhaps using assumptions) to be in Type; i.e., we want as general as possible a partial converse to Theorem 4.3.

Definition 4.5 (Well-formed context) A context is said to be *well-formed* if and only if it has a standard form (4.2) such that the corresponding sequence of assumptions (4.1) is a well-formed environment.

It is easy to show the following result:

Theorem 4.4 *If A is a well-formed context, then*

$$\vdash_{\mathrm{TAC}} A : \mathsf{Type}.^5$$

We would like to show that a context cannot be assigned a type other than Type. To do this, we need to consider places that Type can occur in a deduction. It may

[5] It is, in fact, easy to strengthen Theorem 4.3 to show that if $\vdash_{\mathrm{TAC}} A : \mathsf{Type}$ then A is a well-formed context.

appear that it occurs only on the right of the colon and then only alone. But this is not the case, for consider the following example:

$$\frac{\mathsf{Prop} : \mathsf{Type} \qquad \mathsf{Prop} : \mathsf{Type}}{\lambda x{:}\mathsf{Prop} \,.\, \mathsf{Prop} : (\forall x : \mathsf{Prop})\mathsf{Type}} \; (\forall \mathrm{Ti-v})$$

What we can prove about occurrences of Type requires a definition:

Definition 4.6 (Supercontext) A term A is a *supercontext* if

$$A =_* (\forall x_1 : A_1) \ldots (\forall x_n : A_n)\mathsf{Type}$$

where $(\forall x_1 : A_1) \ldots (\forall x_n : A_n)\mathsf{Prop}$ is a well-formed context. Here, $(\forall x_1 : A_1) \ldots (\forall x_n : A_n)\mathsf{Type}$ is called a *standard form* of A, n is called the *index* of the standard form, and A_1, A_2, ..., A_n are called its *prefix types.*

The remarks after Definition 4.4 about the standard forms of contexts apply equally to those of supercontexts.

The result we want is now as follows:

Theorem 4.5 (a) *If* Γ *is a well-formed environment and if*

$$\Gamma \vdash_{\mathrm{TAC}} M : A,$$

then M *reduces to a term in which there is no occurrence of* Type.

(b) *If* Γ *is a well-formed environment and if*

$$\Gamma \vdash_{\mathrm{TAC}} M : A,$$

and if there is an occurrence of Type *in every term to which* A *reduces, then* A *is a supercontext.*[6]

Proof (a) By induction on the deduction of

$$\Gamma \vdash_{\mathrm{TAC}} M : A.$$

[6]Since it is not, in general, decidable whether or not there is an occurrence of Type in every term to which a given term reduces, it may appear that this theorem involves a nonconstructive use of the law of excluded middle. But in fact, all that is really needed for part (b) is that it is not possible to determine from the deduction that there is a reduction from the term to a term in which Type does not occur, and this can be constructively determined.

(Note that the type of each variable in a well-formed environment satisfies the conditions of the lemma.) In the cases for rules (Eq'κ), the conclusion follows via the Church-Rosser theorem and the fact that no reduction can introduce an occurrence of Type into a term. The remaining cases are easy.

(b) By induction on the deduction of

$$\Gamma \vdash_{\mathrm{TAC}} M : A.$$

The only difficult case is rule ($\forall$e); in this case, suppose that the inference is

$$\frac{M : (\forall x : B)C \qquad N : B}{MN : [N/x]C}$$

If there is an occurrence of Type in every term to which $[N/x]C$ reduces, then by (a) there is an occurrence of Type in every term to which N reduces and hence also in every term to which C reduces. Hence, there is an occurrence of Type in every term to which $(\forall x : B)C$ reduces. Thus, by the induction hypothesis (on the left premise), $(\forall x : B)C$ is a supercontext. It follows that C and hence also $[N/x]C$ are also supercontexts. ■

Define an occurrence of a subterm A of a term M to be the *type of a bound variable* if A is the indicated part of a subterm of the form $\lambda x{:}A\ .\ N$ or $(\forall x : A)B$.

Theorem 4.6 *Let* Γ *be a well-formed environment, and suppose*

$$\Gamma \vdash_{\mathrm{TAC}} M : A,$$

where A is not a supercontext. Then $M =_* N$ for some term N in which every occurrence of the atomic term Prop *is inside the type of a bound variable.*[7]

Proof By induction on the deduction of $\Gamma \vdash_{\mathrm{TAC}} M : A$. ■

Corollary 4.6.1 *If* Γ *is a well-formed environment, and if*

$$\Gamma \vdash_{\mathrm{TAC}} M : A,$$

where A is not a supercontext, then M is not a context.

[7]The condition of the theorem that A is not a supercontext is not constructively decidable. However, all that is really necessary for the theorem is that it not be possible to read from the deduction in question that A is a supercontext, and this can be constructively determined.

Corollary 4.6.2 *If* Γ *is a well-formed environment, and if*

$$\Gamma \vdash_{\mathrm{TAC}} M : \kappa \text{ and } \Gamma \vdash_{\mathrm{TAC}} M : \kappa',$$

then $\kappa \equiv \kappa'$.

Proof Otherwise, we have $\Gamma \vdash_{\mathrm{TAC}} M$: Prop and $\Gamma \vdash_{\mathrm{TAC}} M$: Type, from which we get by Theorem 4.3 that M is a context and from Corollary 4.6.1 that it is not a context. ■

It is not hard to generalize Theorem 4.3 to the following:

Theorem 4.7 *If*

$$\Gamma \vdash_{\mathrm{TAC}} A : B,$$

where B *is a supercontext, then*

$$A =_* \lambda x_1{:}A_1 \,.\, \lambda x_2{:}:A_2 \,.\, \ldots \lambda x_m{:}:A_m \,.\, A', \tag{4.3}$$

where A' *is a context.*

Definition 4.7 (Context Function) A term A satisfying the conclusion of Theorem 4.7 is called a *context function.* If A' is a standard form, then the form on the right of 4.3 is called a *standard form of* A, and its *index* is m plus the index of A'. All of the remarks and conventions regarding standard forms and indices of contexts apply to those of context functions.

Now let us consider the subject-reduction theorem (Theorem 2.1) . In order to prove it, we need a replacement theorem corresponding to Lemma 2.1. Lemma 2.1 is stated in terms of the subject-construction theorem, which is much more complicated to state for TAC than it is for TA, but the part of the lemma corresponding to the subject-construction theorem is not needed for the subject-reduction theorem. Another complication arises from the fact that changes in a term to which a type is assigned may be reflected later in a deduction in the types themselves. However, in the case of the replacement lemma needed for the subject-reduction theorem, a term is replaced by a convertible term, so by rule (Eq″), the later types need not be changed. (See Hindley & Seldin [HS86] Lemma 16.39.) It is sufficient to have the following result (which is called a theorem because it is more substantial than Lemma 2.1):

Theorem 4.8 (Replacement) *Let Γ_1 be any well-formed environment, and let $\mathcal{D}$ be a deduction of*

$$\Gamma_1 \vdash_{\mathrm{TAC}} M : A.$$

Let $V : C$ be any statement in $\mathcal{D}$, let $\mathcal{D}_1$ be that part of $\mathcal{D}$ ending in $V : C$, let $\mathcal{D}_2$ be the rest of $\mathcal{D}$, and let $x_1 : B_1$, $x_2 : B_2$, ..., $x_n : B_n$ be the assumptions of $\mathcal{D}_1$ that are discharged in $\mathcal{D}_2$. Let W be a term such that $W =_* V$ and $FV(W) \subseteq FV(V)$, and suppose that Γ_2 is a well-formed environment in which x_1, x_2, ..., x_n do not occur free. Suppose that $\mathcal{D}_3$ is a deduction of

$$\Gamma_2, x_1 : B_1, \ldots, x_n : B_n \vdash_{\mathrm{TAC}} W : C.$$

Then replacing $\mathcal{D}_1$ by $\mathcal{D}_3$ in $\mathcal{D}$ results in a deduction $\mathcal{D}_4$ of

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} M^* : A,$$

where M^* is obtained from M by replacing appropriate occurrences of V by W.[8]

Proof By induction on the structure of $\mathcal{D}_2$.

Basis: There are two cases.

Case 1. $\mathcal{D}_2$ consists of the single statement $V : C$. Then M is V, M^* is W, and $\mathcal{D}_4$ is just $\mathcal{D}_3$.

Case 2. $\mathcal{D}_2$ consists only of the axiom (P T). Then the replacement is vacuous, $W \equiv V \equiv \mathsf{Prop}$, and $\mathcal{D}_4$ consists only of the axiom (P T).

Induction step: We have the following cases depending on the last inference in $\mathcal{D}_2$.

Case 1. The last inference of $\mathcal{D}_2$ is ($\kappa\kappa'$ Formation).Then A is κ', M is $(\forall x : B)E$, and $\mathcal{D}$ is

$$\frac{\begin{array}{cc} & 1 \\ & [x : B] \\ \mathcal{D}_5 & \mathcal{D}_6(x) \\ B : \kappa & E : \kappa' \end{array}}{(\forall x : B)E : \kappa',} \quad (\kappa\kappa'\text{Formation} - 1)$$

[8] It is difficult to describe exactly the replacements which are required to obtain M^* from M, but it is possible to read the replacement process from the proof. It is worth noting that the part of $\mathcal{D}_4$ which is not included in $\mathcal{D}_3$ has exactlythe same inference rules in the same relative positions as $\mathcal{D}_2$ except perhaps for some inferences by (Eq'κ), (Eq''), or ($\equiv'_\alpha$).

where the occurrence of $V : C$ is either in $\mathcal{D}_5$ or in $\mathcal{D}_6(x)$. By the induction hypothesis, the replacement of $\mathcal{D}_1$ by $\mathcal{D}_3$ in $\mathcal{D}_5$ and $\mathcal{D}_6(x)$ leads to deductions $\mathcal{D}_7$ and $\mathcal{D}_8(x)$ of, respectively,

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} B^* : \kappa$$

and

$$\Gamma_1, \Gamma_2, x : B \vdash_{\mathrm{TAC}} E^* : \kappa'$$

for appropriate B^* and E^*. Since $V =_* W, B^* =_* B$, and so $\mathcal{D}_4$ is as follows:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_7\\ B^* : \kappa\end{array}}{B : \kappa}\ (\mathrm{Eq}'\kappa) \qquad \begin{array}{c}1\\ [x : B]\\ \mathcal{D}_8(x)\\ E^* : \kappa'\end{array}}{(\forall x : B)E^* : \kappa'.}\ (\kappa\kappa'\text{Formation} - 1)$$

Case 2. The last inference of $\mathcal{D}$ is by $(Eq'\kappa)$. Then A is κ and $\mathcal{D}$ is

$$\dfrac{\begin{array}{c}\mathcal{D}_5\\ N : \kappa\end{array}}{M : \kappa,}\ (\mathrm{Eq}'\kappa)$$

where $N =_* M$. By the induction hypothesis, the replacement of $\mathcal{D}_1$ by $\mathcal{D}_3$ in $\mathcal{D}_5$ leads to a deduction $\mathcal{D}_6$ of

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} N^* : \kappa$$

for an appropriate N^*. Since $N^* =_* N =_* M$, we can take $M^* \equiv M$, and then $\mathcal{D}_4$ is obtained from $\mathcal{D}_6$ by an inference by $(Eq'\kappa)$.

Case 3. The last inference of $\mathcal{D}$ is by $(\forall \mathrm{e})$. Then M is $M_1 M_2, A$ is $[M_2/x]A'$, and $\mathcal{D}$ is

$$\dfrac{\begin{array}{cc}\mathcal{D}_5 & \mathcal{D}_6\\ M_1 : (\forall x : B)A' & M_2 : B\end{array}}{M_1 M_2 : [M_2/x]A'.}\ (\forall \alpha\ \mathrm{e})$$

By the induction hypothesis, the replacement of $\mathcal{D}_1$ by $\mathcal{D}_3$ in $\mathcal{D}_5$ and $\mathcal{D}_6$ leads to deductions $\mathcal{D}_7$ and $\mathcal{D}_8$ of

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} M_1^* : (\forall x : B)A'$$

and

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} M_2^* : B$$

for appropriate M_1^* and M_2^*. Furthermore, $M_2^* =_* M_2$. Hence, $\mathcal{D}_4$ is

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D}_7 & \mathcal{D}_6 \\ M_1^* : (\forall x : B)A' & M_2^* : B\end{array}}{M_1^* M_2^* : [M_2^*/x]A'}\ (\forall\alpha\,\mathrm{e})}{M_1^* M_2^* : [M_2/x]A'.}\ (\mathrm{Eq}'')$$

Case 4. The last inference of $\mathcal{D}$ is by $(\forall\kappa\mathrm{i})$. Then A is $(\forall x : B)E$, M is $\lambda x : B\,.\,N$, and $\mathcal{D}$ is

$$\dfrac{\begin{array}{cc} 1 & \\ [x : B] & \\ \mathcal{D}_5(x) & \mathcal{D}_6 \\ N : E & B : \kappa\end{array}}{\lambda x{:}B\,.\,N : (\forall x : B)E.}\ (\forall\kappa\mathrm{i} - 1)$$

By the induction hypothesis, the replacement of $\mathcal{D}_1$ by $\mathcal{D}_3$ in $\mathcal{D}_5(x)$ and $\mathcal{D}_6$ leads to deductions $\mathcal{D}_7(x)$ and $\mathcal{D}_8$ of

$$\Gamma_1, \Gamma_2, x : B \vdash_{\mathrm{TAC}} N^* : E$$

and

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} B^* : \kappa$$

for appropriate N^* and B^*, where $B^* =_* B$. Then $\mathcal{D}_4$ is as follows:

$$\dfrac{\dfrac{\begin{array}{cc} 1 & \\ [x : B] & \mathcal{D}_8 \\ \mathcal{D}_7(x) & B^* : \kappa \\ N^* : E & \end{array}}{B : \kappa}\ (\mathrm{Eq}'\kappa)}{\lambda x{:}: B\,.\,N^* : (\forall x : B)E.}\ (\forall\kappa\mathrm{i} - 1)$$

Case 5. The last inference of $\mathcal{D}$ is by (Eq″). Then $\mathcal{D}$ is

$$\dfrac{\begin{array}{c}\mathcal{D}_5\\ M : B\end{array}}{M : A,} \ (\mathrm{Eq}'')$$

where $A =_* B$. By the induction hypothesis, the replacement of $\mathcal{D}_1$ by $\mathcal{D}_3$ in $\mathcal{D}_5$ leads to a deduction $\mathcal{D}_6$ of

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} M^* : B$$

for appropriate M^*, and $\mathcal{D}_4$ is obtained by adding an inference by (Eq″) at the end.

Case 6. The last inference in $\mathcal{D}$ is by $(\equiv'_\alpha)$. Then $\mathcal{D}$ is

$$\dfrac{\begin{array}{c}\mathcal{D}_5\\ N : A\end{array}}{M : A,} \ (\equiv'_\alpha)$$

where M is obtained from N by changes of bound variables. By the induction hypothesis, the replacement of $\mathcal{D}_1$ by $\mathcal{D}_3$ in $\mathcal{D}_5$ leads to a deduction $\mathcal{D}_6$ of

$$\Gamma_1, \Gamma_2 \vdash_{\mathrm{TAC}} N^* : A$$

for appropriate N^*. Since $FV(W) \subseteq FV(V)$, the changes of bound variables which occur in passing from N to M will take N^* to the desired M^*, and so $\mathcal{D}_4$ can be obtained from $\mathcal{D}_6$ by adding an inference by $(\equiv'_\alpha)$. ■

We can use this theorem to prove the subject-reduction theorem the same way that Lemma 16.39 of Hindley & Seldin [HS86] is used to prove Theorem 16.41:

Theorem 4.9 (Subject-reduction theorem) *Let Γ be a well-formed environment. If*

$$\Gamma \vdash_{\mathrm{TAC}} M : A$$

and $M N$, then

$$\Gamma \vdash_{\mathrm{TAC}} N : A.$$

(See also the proof of Hindley & Seldin [HS86] Theorem 15.17).

As in Hindley & Seldin [HS86] §16D2, the subject-reduction theorem is related to the normalization theorem. In particular, it tells us the result of performing a reduction step on a valid deduction is another valid deduction. The reduction steps that interest us are the following:

κ reductions. A deduction of the form

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}1\\ [x:A]\\ \mathcal{D}_1(x)\\ M:B\end{array}\qquad\begin{array}{c}\\ \\ \mathcal{D}_2\\ A:\kappa\end{array}}{\lambda x{:}A\,.\,M:(\forall x:A)B}\,(\forall\kappa\mathrm{i}-1)}{\lambda x{:}A\,.\,M:(\forall x:C)B}\,(\mathrm{Eq}'')\qquad\begin{array}{c}\mathcal{D}_3\\ N:C\end{array}}{(\lambda x{:}A:M)N:[N/x]B}\,(\forall\alpha\ \mathrm{e})}{\mathcal{D}_4}$$

reduces to

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_3\\ N:C\end{array}}{N:A}\,(\mathrm{Eq}'')}{\begin{array}{c}\mathcal{D}_1(N)\\ [n/x]M:[N/x]B\\ \mathcal{D}_4',\end{array}}$$

where $\mathcal{D}_4'$ is obtained from $\mathcal{D}_4$ by replacing appropriate occurrences of $(\lambda x{:}a\,.\,M)N$ by $[N/x]M$ according to Theorem 4.8.

Here, the formula $\lambda x{:}a\,.\,M:(\forall x:C)B$ *the cut* formula of the reduction step. A reduction is a (possibly empty) sequence of replacements using these reduction steps.

A special case of a κ reduction step is a *context-reduction step or c-reduction step* in which B is a context or a supercontext. A *context-reduction or c-reduction* is a reduction in which each reduction step is a c-reduction step. A deduction will be said to be *context-normal, or c-normal* if it contains no cut formulas for c-reduction

steps. It turns out to be easy to prove that every deduction can be reduced to a c-normal deduction using the notion of the *degree* of a term, and that this partial normalization result is important in proving the full normalization theorem.

Definition 4.8 (Degree of a term) Let A be a term such that there is a step $M : A$ in a deduction in TAC. Then the *degree* of A relative to the deduction is defined as follows:

(a) if A is not a context or a supercontext, then the degree of A is 0;

(b) the degrees of Prop and Type are 1;

(c) the degree of $(\forall x : A)B$ is one more than the maximum of the degrees of A and B; and

(d) if $A =_* B$, then the degree of A is equal to the degree of B.

Since only contexts and supercontexts have nonzero degrees, the definition of a context is enough to guarantee that the degree of a term relative to a deduction is well defined.

Remark Since it is not possible to decide mechanically for a given term whether or not it is a context or a supercontext, it may appear that this definition uses the law of the excluded middle, which is invalid in constructive logic, to define the degree of a term. But this is not really the case; for in calculating the degree of a given context or supercontext, it is only necessary to calculate the degree of terms A which are either Prop or Type or for which there is a step in the deduction of the form A : Type or A : Prop, and then the degree of A can be determined by which of these situations occurs. (It is impossible to have more than one by Theorems 4.3, 4.4, 4.5 and 4.6, and it is possible to determine mechanically which occurs.)

Note that the degree of a term relative to a deduction is invariant of β-conversion.

Theorem 4.10 *Every deduction in TAC with conclusion $M : A$ can be reduced to a c-normal deduction with the same undischarged assumptions and with conclusion $N : A$, where $M N$.*

Proof Let the degree of a cut formula be the degree of its type with respect to the deduction. Note that if a cut formula is removed by a reduction step, the degree of another cut formula which had lower degree before the reduction step and which occurs in the deduction after the reduction is unchanged. Let the *index* of a deduction be the pair $\langle d, n \rangle$, where d is the maximum degree of any cut formula in the deduction and n is the number of cut formulas in the deduction with degree d. If the pairs are ordered as in the proof of Theorem 1.2, and if reduction steps

are carried out in the same order (the cut formula has degree d, and there is no cut formula with degree d in $\mathcal{D}_3$), then an argument like that of the proof of Theorem 1.2 shows that every deduction can be reduced to a deduction with no cut formulas. It should be clear from the nature of the reduction steps that a reduction changes only the term to the left of the colon in any formula by carrying out a sequence of contractions. ■

Definition 4.9 The term N of Theorem 4.10 will be called a *c-normal form* of M.

In terms of this definition, Theorem 4.10 says that every term to which a type is assigned by TAC has a c-normal form.

This partial normalization result is important for the full normalization theorem because it gives us some useful information about terms A for which it is possible to prove $\Gamma \vdash_{\mathrm{TAC}} A : \mathsf{Prop}$. To obtain this information, we need the following lemmas:

Lemma 4.1 *Let $\mathcal{D}$ be a c-normal deduction of*

$$\Gamma \vdash_{\mathrm{TAC}} A : \mathsf{Prop},$$

where Γ is a well-formed environment. Then either $A =_* (\forall x : B)C$ for some terms B and C and some variable x which does not occur free in Γ, or $A =_* xM_1M_2 \ldots M_p$ for some variable x, some natural number p (which may be 0), and some terms $M_1, M_2, \ldots, M_p$, and furthermore, it can be decided constructively which of these alternatives holds.

Proof Consider the last inference in $\mathcal{D}$ which is not by (Eq″), (Eq′P), or ($\equiv'_\alpha$). This inference cannot be by ($\forall\kappa$i) since the type of the conclusion is an atomic constant, so the only remaining possible rules are (κP Formation) and ($\forall$e). Which of these rules actually occurs can be decided constructively (by inspection of the deduction).

If the inference is by (κP Formation), then there are terms B and C and a variable x which does not occur free in Γ such that $A =_* (\forall x : B)C$.

If the inference is by ($\forall$e), then consider the left branch of the deduction. As we travel up that branch from the bottom, the only inferences we find are by ($\forall$e), (Eq″), ($\equiv'_\alpha$), and perhaps (Eq′P) at the very bottom. This means that the formula at the top of the left branch must be an undischarged assumption, and it must therefore be in Γ. It follows that this statement must have the form $x : B$, where $B =_* (\forall x : C_1)\ldots(\forall x : C_p)\mathsf{Prop}$ for some natural number p (which may be 0). Then we must have $A =_* xM_1 \ldots M_p$ for some terms $M_1, \ldots, M_p$. ■

Definition 4.10 (Simple and compound deductions) If $\mathcal{D}$ is a deduction as in Lemma 4.1, then it will be called *compound* if the first case of the lemma holds and *simple* if the second case holds. If A is a term such that $A : \mathsf{Prop}$ is the conclusion of such a deduction $\mathcal{D}$, then A will be *simple* [*compound*] if $\mathcal{D}$ is simple [compound].

Lemma 4.2 *If there is a deduction of*

$$\Gamma \vdash_{\mathrm{TAC}} A : \mathsf{Prop},$$

then there is a c-normal deduction of it.

Proof Let $\mathcal{D}$ be the given deduction. By Theorem 4.10 there is a c-normal deduction of

$$\Gamma \vdash_{\mathrm{TAC}} B : \mathsf{Prop},$$

where AB. By adding one inference by (Eq'P) at the end, we get the desired c-normal deduction of

$$\Gamma \vdash_{\mathrm{TAC}} A : \mathsf{Prop}.$$

■

By Lemma 4.2 and Definition 4.10, every type in Prop (with respect to a given well-formed environment) is either simple or compound, and it is possible to decide constructively which it is. Furthermore, the compound types are formed by repeated use of the operation $\forall$ from the simple types and Prop. Note that the contexts are formed in more or less the same way.

Lemma 4.3 *If $\mathcal{D}$ is a deduction of*

$$\Gamma \vdash_{\mathrm{TAC}} (\forall x : A)B : \mathsf{Prop},$$

where x does not occur free in Γ or in A and where Γ is a well-formed environment, then there is a deduction $\mathcal{D}'$ of

$$\Gamma, x : A \vdash_{\mathrm{TAC}} B : \mathsf{Prop}.$$

Furthermore, the c-normal deduction to which $\mathcal{D}'$ reduces has fewer inferences by rules other than (Eq''), (Eq'κ), *and* ($\equiv'_\alpha$) *than the c-normal deduction to which $\mathcal{D}$ reduces.*

Proof This follows from Lemmas 4.1 and 4.2. ■

Theorem 4.11 *If*

$$\Gamma \vdash_{\mathrm{TAC}} M : A,$$

where Γ *is a well-formed environment and* A *is not a supercontext, then*

$$\Gamma \vdash_{\mathrm{TAC}} A : \mathsf{Type}$$

or

$$\Gamma \vdash_{\mathrm{TAC}} A : \mathsf{Prop}.$$

Proof By induction on the length of the deduction $\mathcal{D}$ with the conclusion $M : A$. The only difficult case is that in which the last inference of $\mathcal{D}$ is by rule ($\forall$e). Then $M \equiv PN, A \equiv [N/x]C$, and $\mathcal{D}$ has the form

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ P : (\forall x : B)C & N : B\end{array}}{PN : [n/x]C.} \ (\forall\alpha\ \mathrm{e})$$

By the induction hypothesis,

$$\Gamma \vdash_{\mathrm{TAC}} (\forall x : B)C : \kappa, \tag{4.4}$$

and

$$\Gamma \vdash_{\mathrm{TAC}} B : \kappa', \tag{4.5}$$

If we have $\kappa \equiv \mathsf{Type}$, then 4.4 must be the conclusion of either (κ''TFormation), the premises being 4.5 and

$$\Gamma, x : B \vdash_{\mathrm{TAC}} C : \mathsf{Type}.$$

The conclusion then follows placing $\mathcal{D}_2$ over each occurrence of the assumption $x : B$. If $\kappa \equiv \mathsf{Prop}$, we use Lemma 4.3 to carry out a similar argument using one of the rules rules (κP Formation). ■

Lemmas 4.1 and 4.2 give us a structure on the types in Prop. It is interesting to note that the other types have exactly the same structure. By Theorem 4.11, every type is in Prop, in Type, or is a supercontext. It is clear from the definition that supercontexts have this structure, and Theorem 4.3 tells us that the same is true for contexts. What all of this means is that types are built up from Type, Prop, and the simple types by the operation forming $(\forall x : A)B$.

Theorems 4.3, 4.4 and 4.11 and Corollary 4.6.1 allow us to classify all formulas which can be deduced from well-formed environments:

Definition 4.11 (Classification of formulas) A formula $M : A$ is called:
(a) a *context function* if A is a supercontext;
(b) a *context* if $A =_* \mathsf{Type}$;
(c) a *proposition function* if A is a context;
(d) a *proposition* if $A =_* \mathsf{Prop}$; and
(e) a *proof* if A is neither a context nor a supercontext.
A deduction whose undischarged assumptions form a well-formed environment is classified according to its last formulas.

This classification shows the connection between TAC and the formulas-as-types isomorphism.

We would like to extend this classification to the terms M (at least relative to a given well-formed environment). In other words, we modify Definition 4.11 as follows:

Definition 4.12 (Classification of terms) A term M is called:
(a) a Γ-*context function* if there is a supercontext A such that $\Gamma \vdash_{\mathrm{TAC}} M : A$;
(b) a Γ-*context* if $\Gamma \vdash_{\mathrm{TAC}} M : \mathsf{Type}$;
(c) a Γ-*proposition function* if there is a context A such that $\Gamma \vdash_{\mathrm{TAC}} M : A$;
(d) a Γ-*proposition* if $\Gamma \vdash_{\mathrm{TAC}} M : \mathsf{Prop}$; and
(e) a Γ-*proof* if there is a term A which is neither a context nor a supercontext such that $\Gamma \vdash_{\mathrm{TAC}} M : A$.

We have already proved (Corollary 4.6.1) that no term is both a Γ-context function and a Γ-proposition function or both a Γ-context function and a Γ-proof. To complete the proof that this a classification is exclusive, we need the following result.

Theorem 4.12 *If Γ is a well-formed environment, and if*

$$\Gamma \vdash_{\mathrm{TAC}} M : A \text{ and } \Gamma \vdash_{\mathrm{TAC}} M' : B,$$

are both derivable, where M and M' differ only by changes of bound variables, then $A =_* B$.

Proof By induction on the lengths of the two deductions, $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively.

Case 1. The last inference in $\mathcal{D}_1$ is by (Eq″). Assume that the left premise is $M : A'$. By the induction hypothesis, $A' =_* B$. But $A =_* A'$, and so $A =_* B$.

Case 2. The last inference in $\mathcal{D}_2$ is by (Eq″). Symmetric to Case 1.

Case 3. The last inference in neither $\mathcal{D}_1$ nor $\mathcal{D}_2$ is by (Eq″).

Subcase 3.1. $\mathcal{D}_1$ consists of the axiom. Then M is Prop and A is Type. Then either $\mathcal{D}_2$ is also the axiom, in which case B is Type and we are finished, or else the last inference in $\mathcal{D}_2$ is by rule (Eq′κ), in which case κ is Type by Corollary 4.6.1.

Subcase 3.2. The last inference of $\mathcal{D}_1$ is by ($\kappa\kappa'$Formation). Then B is κ' by Corollary 4.6.2.

Subcase 3.3. The last inference of $\mathcal{D}_1$ is by (Eq′κ). Then by Corollary 4.6.2, B is κ.

Subcase 3.4. The last inference of $\mathcal{D}_1$ is by ($\forall\alpha$ e). Then the last inference of $\mathcal{D}_2$ is either ($\forall\alpha$ e) or (Eq′κ). If it is (Eq′κ), then the theorem follows by Corollary 4.6.2 Otherwise, M is NP, M' is $N'P'$ (where N' and P' differ from N and P only by changes in bound variables, A is $[P/x]A'$, B is $[P/x]B'$, $\mathcal{D}_1$ is

$$\frac{\begin{array}{cc}\mathcal{D}_{11} & \mathcal{D}_{12}\\ N : (\forall x : C)A' & P : C\end{array}}{NP : [P/x]A',}\ (\forall\alpha\ \text{e})$$

and $\mathcal{D}_2$ is

$$\frac{\begin{array}{cc}\mathcal{D}_{21} & \mathcal{D}_{22}\\ N : (\forall x : D)B' & P : B\end{array}}{NP : [P/x]B'.}\ (\forall\alpha\ \text{e})$$

By the induction hypothesis, $C =_* D$ and $(\forall x : C)A' =_* (\forall x : D)B'$. It follows that $A' = *B'$, and hence $A =_* B$.

Subcase 3.5. The last inference in $\mathcal{D}_1$ is by ($\forall\kappa$i). Then the last inference in $\mathcal{D}_2$ is by ($\forall\kappa$i), M is $\lambda x{:}C\ .\ N$, M' is $\lambda x{:}C\ .\ N'$ where N and N' differ by changes in bound variables, A is $(\forall x : C)A'$, and B is $(\forall x : C)B'$. (There is no loss of generality in assuming that the indicated bound variable is x in both M and M' because if the bound variables are different a minor modification of $\mathcal{D}_2$ will make them the

same.) Furthermore, $\mathcal{D}_1$ is

$$\dfrac{\begin{array}{c}1\\ [x:C]\\ \mathcal{D}_{11}\\ N:A' \qquad C:\kappa\end{array}}{\lambda x{:}C\,.\,N:(\forall x:C)A'}\ (\forall\kappa\mathrm{i}-1)$$

and $\mathcal{D}_2$ is

$$\dfrac{\begin{array}{c}1\\ [x:C]\\ \mathcal{D}_{21}\\ N':B' \qquad C:\kappa'\end{array}}{\lambda x{:}C\,.\,N':(\forall x:C)B'.}\ (\forall\kappa'\mathrm{i}-1)$$

By the induction hypothesis, $A' =_* B'$, and it clearly follows that $A =_* B$.

Subcase 3.6. The last inference in $\mathcal{D}_1$ is by $(\equiv'_\alpha)$. This case is trivial. ■

Corollary 4.12.1 *For any well-formed environment* Γ, *no term is both a* Γ*-proposition function and a* Γ*-proof.*

Proof Suppose M is both a Γ-proposition function and a Γ-proof. Then there is a Γ-proposition B and a Γ-context C such that

$$\Gamma \vdash_{\mathrm{TAC}} M:B \text{ and } \Gamma \vdash_{\mathrm{TAC}} M:C.$$

Hence,

$$\Gamma \vdash_{\mathrm{TAC}} B:\mathsf{Prop} \text{ and } \Gamma \vdash_{\mathrm{TAC}} C:\mathsf{Type}.$$

By the theorem, $B =_* C$. Hence, by the Church-Rosser Theorem, there is a term D to which both B and C reduce which can be proved on the basis of Γ to be in both Prop and Type, contradicting Corollary 4.6.2. ■

Theorem 4.10 gives us the following characterization of Γ-proposition functions:

Theorem 4.13 *If* Γ *is a well-formed environment, and if* A *is a* Γ*-proposition function which is not a proposition, then either each c-normal form of* A *has the form* $\lambda x{:}B:C$, *in which case the type assigned to* A *by* Γ *converts to* $(\forall x:B)F$, *where* F *is a context, or each c-normal form of* A *has the form* $xM_1\ldots M_n$.

Proof By hypothesis, there is a c-normal deduction of

$$\Gamma \vdash_{\mathrm{TAC}} D : (\forall x : B)E,$$

where AD, which is a c-normal form of it, and B is a context. Except for (Eq″) and ($\equiv_\alpha$), which make no difference, the last inference in this c-normal deduction must be ($\forall\kappa$i) or ($\forall\alpha$ e). If it is ($\forall\kappa$i), we are done. If it is ($\forall\alpha$ e), then proceed up the left branch to the first formula which is not the conclusion of an inference by ($\rightarrow$ e) or ($\forall\alpha$ e). Since the deduction is c-normal and since Γ is a context, this formula is not the conclusion of an inference by ($\forall\kappa$i). Hence, it is an assumption, and D has the form $xM_1 \ldots M_n$, as desired. (That all c-normal forms of A are of the same kind follows by the Church-Rosser Theorem.) ■

By iterating the theorem, and, if necessary, replacing terms M by $\lambda y_i{:}B_i \,.\, My_i$, where y_i is not free in M, we can prove the following corollary:

Corollary 4.13.1 *Under the hypotheses of the theorem, if*

$$\Gamma \vdash_{\mathrm{TAC}} A : (\forall x_1 : B_1)\ldots(\forall x_n : B_n)\mathsf{Prop},$$

then either $A =_* \lambda x_1{:}B_1 \,.\, \ldots \lambda x_n{:}B_n \,.\, A'$, *where* A' *is a* Γ*-context, or else every c-normal form of* A *has the form* $xM_1 \ldots M_n$.

Remark It is worth pointing out that, as we have formulated TAC, there is nothing to exclude making an assumption of the form $x : A$, where A is a supercontext. We have not considered such assumptions so far, and the early formulations of TAC excluded them. But they do no harm, since the rules of the system prevent the discharge of any such assumption. Furthermore, they will turn out to be useful in practice, since undischarged variables may be thought of as new constants added to the system. But if such assumptions are allowed, then it is no longer true that anything that can be proved to be in Type is a context in the sense of Definition 4.4; it might convert instead to

$$(\forall x_1 : A_1)\ldots(\forall x_n : A_n)xB_1 \ldots B_m.$$

If we allow such terms to be contexts in a generalized sense, then different assumptions can result in the same formula having different classifications according to Definition 4.11. For example, let Γ_1 be x : Type and let Γ_2 be x : Prop; then $y : x$ is a Γ_1-proposition and a Γ_2-proof. Furthermore, the definition of well-formed environment (Definition 4.3) would have to be modified to allow any of the A_i to be a supercontext. (Definition 4.5, of a well-formed context, would then have to differ

from Definition 4.3, since none of the A_i of a standard form of a well-formed context can convert to a supercontext.) In Definition 4.8, it is necessary to specify that the rank of $xB_1 \ldots B_m$ is 1 if $x : (\forall x_1 : A_1) \ldots (\forall x_m : A_m)\mathsf{Type}$ is assumed in the deduction. In connection with Definition 4.10, a term of the form $xB_1 \ldots B_m$, where $x : (\forall x_1 : A_1) \ldots (\forall x_m : A_m)\mathsf{Type}$ assumed in the deduction, will be called a simple generalized context. Finally, it is important to specify that no substitutions be made for variables assumed to be in supercontexts; they must behave like constants. In what follows, we shall assume that these modifications have been made.

4.3 The strong normalization theorem.

It might appear that to prove the normalization theorem it is sufficient to combine Theorem 4.10 with a similar result for reduction steps whose cut formulas are not propositions. But this fails to work, for on the one hand, such a reduction step may require that a type of arbitrary complexity be substituted for a variable that is part of an assumption that is also a sentence, and on the other hand, a reduction step whose cut formula is a proof may introduce a new cut formula which is a proposition and whose type is a context of arbitrarily high degree.

On the other hand, Theorem 4.10 is of help in proving normalization, for it shows (via Lemma 4.3) that the types which are proved to be in Prop can be formed from the simple types and Prop by $\forall$ in much the same way that the types of TAP are formed from type variables by the type constructors. This turns out to make it possible to adapt a proof of normalization for TAP to TAC. The proof we have chosen to adapt is a proof of *strong normalization* due to Stenlund [Ste72] §5.6. However, the proof needs to be modified in much the way that the proof of [Mar71a] is modified in [Mar73].

Convention Let $\mathcal{D}$ be a deduction whose conclusion is $M : A$, where $A =_* (\forall x_1 : A_1)\ldots(\forall x_n : A_n)B$, and for $i = 1,\ldots,n$, let $\mathcal{D}_1$ be a deduction with conclusion $M_i : A_i'$, where

$$A_i' \equiv [M_1/x_1,\ldots,M_{i-1}/x_{i-1}]A_i.$$

Then

$$\begin{array}{c}\mathcal{D}\\ M : A\\ \{\mathcal{D}_1,\ldots,\mathcal{D}_n\}\end{array}$$

will denote the deduction

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}}{M : A}}{M : (\forall x_1 : A_1)\ldots(\forall x_n : A_n)B}\,(\mathrm{Eq}'') \qquad \mathcal{D}_1 \\ M_1 : A_1'}{MM_1 : [M_1/x_1](\forall x_2 : A_2)\ldots(\forall x_n : A_n)B}\,(\forall\,\mathrm{e}) \\ \vdots \qquad\qquad \mathcal{D}_n \\ MM_1\ldots M_{n-1} : (\forall x_n : A_n')B' \qquad\qquad M_n : A_n'}{MM_1\ldots M_n : B''},\,(\forall\,\mathrm{e})$$

where $B' \equiv [M_1/x_1, \ldots, M_{n-1}/x_{n-1}]B$ and $B'' \equiv [M_1/x_1, \ldots, M_n/x_n]B$. (If $n = 0$, then it will denote $\mathcal{D}$ itself.)

Definition 4.13 (Type of a deduction) If $\mathcal{D}$ is a deduction whose conclusion is $M : A$, then A is called the *type* of $\mathcal{D}$.

Definition 4.14 (Strongly normal deduction) A deduction $\mathcal{D}$ is said to be *strongly normal (SN)* if every reduction starting with $\mathcal{D}$ terminates in a normal deduction.

Our aim is to prove that every deduction is SN.

Remark In the proof, we will be making important use of the classifications in Definition 4.11. We will also be discussing a number of deductions at the same time. It will be important that each formula in each deduction be classified the same way in any other deduction under consideration. For this purpose we will need to know that the well-formed environments of different deductions are all consistent in that none of them have assumptions assigning different types to the same variable. To ensure this consistency, we will assume that we are starting with a generalized well-formed environment Γ_0 that is an infinite set rather than a finite sequence of assumptions. All well-formed environments actually considered will draw their assumptions from Γ_0, and no variable will be assigned more than one type in Γ_0. Furthermore, we shall assume that any finite subset of Γ_0 can be extended to a larger finite subset of Γ_0 whose elements can be ordered in such a way that it is a well-formed environment. For any deduction under consideration,

we shall assume that its discharged assumptions belong to Γ_0; such a deduction will be called Γ_0-acceptable. A term which is the type of a Γ_0-acceptable deduction will be called a Γ_0-type. We shall assume that any term is a Γ_0-type which can be built up from Prop, Type, and the simple types and simple generalized contexts obtainable from assumptions in Γ_0. (This assumption is easy to satisfy; if we start with a candidate for Γ_0 for which it is not true, we extend it with new assumptions (for new variables), and we keep doing this until there are enough assumptions.) A Γ_0-proposition variable of type A, where A is a context, is a variable x such that $x : A$ is in Γ_0. And finally, a Γ_0-term of type A is a term M such that $M : A$ is provable from assumptions in Γ_0.

Definition 4.15 (Ground type set) A set S of Γ_0-acceptable deductions is a grounded type set (ground) if the following three conditions are satisfied:
(a) Every deduction in S is SN;
(b) If $\mathcal{D}_1(N)$ is a part of a deduction obtained from a deduction

$$\begin{array}{c} x : A \\ \mathcal{D}_1(x) \\ M : B \end{array}$$

by substituting N for x, if $\mathcal{D}_3$ is SN, and if

$$\dfrac{\begin{array}{c}\mathcal{D}_3 \\ N : C\end{array}}{\begin{array}{c} N : A \\ \mathcal{D}_1(N) \\ [N/x]M : [N/x]B \\ \mathcal{D}_1', \ldots, \mathcal{D}_n' \end{array}} \quad (\mathrm{Eq}'')$$

is in S, then

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}1\\ [x:A]\\ \mathcal{D}_1(x)\\ M:B\end{array}\qquad\begin{array}{c}\\ \\ \mathcal{D}_2\\ A:\kappa\end{array}}{\lambda x{:}A\,.\,M:(\forall x:A)B}\,(\forall\kappa\ \mathrm{i}-1)}{\lambda x{:}A\,.\,M:(\forall x:C)B}\,(\mathrm{Eq}'')\qquad\begin{array}{c}\\ \\ \mathcal{D}_3\\ N:C\end{array}}{(\lambda x{:}A.M)N:[N/x]B}\,(\forall\,\mathrm{e})$$

$$\{\mathcal{D}_1',\ldots,\mathcal{D}_n'\}$$

is also in S; and
(c) If $\mathcal{D}_1,\ldots,\mathcal{D}_n$ are SN, and if

$$\begin{array}{c}x:A\\ \{\mathcal{D}_1,\ldots,\mathcal{D}_n\}\end{array}$$

is a Γ_0-acceptable deduction, then it is in S. A ground in which all of the deductions have a given type A will be called a *ground of type A*.

Examples The set of all SN Γ_0-acceptable deductions is a ground. This ground will be called SN. If A is a Γ_0-type, then the set of all Γ_0-acceptable deductions of type A is a ground of type A; it is called SN_A.

Definition 4.16 (Proposition term) A *proposition term* is a term A such that $A:B$ is a proposition. A proposition term which is also a variable is a *proposition variable*. If $B =_* (\forall x_1:B_1)\ldots(\forall x_n:B_n)$ Prop, then terms $M_1,\ldots,M_n$ such that for $i=1,2,\ldots,n$, $M_i:[B_1/x_1,\ldots,B_{i-1}/x_{i-1}]B_i$ can be proved from hypotheses from Γ_0, will be called *argument terms* of A. If $n=0$, then the term [variable] is called a *sentence term* [*sentence variable*]. (Note that if A is a proposition term and $M_1,\ldots,M_n$ are argument terms of A, then $AM_1\ldots M_n$: Prop can be proved from assumptions in Γ_0.)

For the next definition, we need to recall what we know about Γ_0-types. We know that any such type (except a supercontext) can be proved (from assumptions

in Γ_0) to be in Prop or in Type, and that a deduction proving that A is in Prop or Type which has been transformed by Theorem 2.5 can end with an inference by rule (Eq'κ). If we take such a deduction which is c-normal and delete this last inference, we get what we might call a *standard form of* A, to which A converts. If we add to these standard forms the standard forms of the supercontexts, then this standard form will either be Prop, Type, a simple type, a simple generalized context, or else will have the form $(\forall x : B)C$. When we speak of making a definition by induction on the structure of a type, we will mean by induction on the number of occurrences of $\forall$ in its standard form. This mirrors the construction of the type from Prop and the simple types by the universal type-forming operator. We can indicate this induction by the following definition:

Definition 4.17 (Rank of a Γ_0-type) The *rank* of a Γ_0-type A, $\mathrm{rk}(A)$, is defined as follows:

(a) if A is a simple type or a simple generalized context, $\mathrm{rk}(A) = 0$;

(b) $\mathrm{rk}(\mathsf{Prop}) = \mathrm{rk}(\mathsf{Type}) = 0$;

(c) $\mathrm{rk}((\forall x : A)B) = \mathrm{rk}(A) + \mathrm{rk}(B) + 1$; and

(d) if $A =_* B$, then $\mathrm{rk}(A) = \mathrm{rk}(B)$.

Definition 4.18 (Computability predicate) Let M be a Γ_0-term of type A. By induction on $\mathrm{rk}(A)$, a *computability predicate of type* M, denoted $p[M]$ is defined as follows:

(a) if A is not a context, then $p[M] \equiv M$;

(b) if $A =_*$ Prop or Type, then $p[M]$ is a ground of type M; and

(c) if $A =_* (\forall x_1 : A_1)\ldots(\forall x_n : A_n)\mathsf{Prop}$, then $p[M]$ is a function whose arguments are computability predicates $\mathrm{p}[M_1], \ldots, \mathrm{p}[M_n]$ of types $M_1, \ldots, M_n$, where each M_i is a Γ_0-term of type A_i, and whose value is a ground of type $MM_1 \ldots M_n$.

For the next definition, we need to proceed by a kind of induction on the structure of a term. For this induction, we need to note that if a term A is not a Γ_0-proof, then it is a Γ_0-proposition function, a Γ_0-context function, or a supercontext. Thus, if A is not a Γ_0-proof, then it converts to Prop, Type, a Γ_0-simple type, a Γ_0-simple generalized context, $(\forall x : B)C$ (where B is neither a supercontext nor a proof and where C is not a proof), or $\lambda x : B \ . \ C$ (where B is neither a supercontext nor a proof and where C is neither a supercontext nor a proof). Here B and C are essentially simpler than A; furthermore, if A converts to a simple type $xM_1 \ldots M_n$, then each M_i is essentially simpler than A. This justifies the following definition by induction on the "structure of A".

Definition 4.19 (Computability object) Let $A(x_1,\ldots,x_n)$ be a term all of whose free variables which are not assigned to supercontexts in Γ_0 occur in the list $x_1,\ldots,x_n$. Let $A_1,\ldots,A_n$ be Γ_0-terms of the types of $x_1,\ldots,x_n$ respectively. Let $p[A_1],\ldots,p[A_n]$ be an assignment of computability functions to the terms $A_1,\ldots,A_n$. Relative to this assignment we shall define by induction on the structure of $A(x_1,\ldots,x_n)$ a *computability object* $\mathsf{C}[A(x_1,\ldots,x_n)]\,(p[A_1],\ldots,p[A_n])$, which will contain deductions of type $A(A_1,\ldots,A_n)$ if $A(x_1,\ldots,x_n)$ is a Γ_0-type. To simplify the notation, we let $\boldsymbol{x}$ be the sequence $x_1,\ldots,x_n$, $\boldsymbol{A}$ the sequence $A_1,\ldots,A_n$, and $\boldsymbol{p}[\boldsymbol{A}]$ be the sequence $p[A_1],\ldots,p[A_n]$.

(a) if $A(\boldsymbol{x})$ is a Γ_0-proof, then $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}])$ is the term $A(\boldsymbol{A})$ itself;

(b) if $A(\boldsymbol{x}) =_*$ Prop, Type, or a Γ_0-simple generalized context, then $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}]) = SN_{A(\boldsymbol{A})}$;

(c) if $A(\boldsymbol{x}) =_* x_i M_1(\boldsymbol{x})\ldots M_m(\boldsymbol{x})$ and is neither a Γ_0-proof nor a Γ_0-simple generalized context, then $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}])$ is $p[A_i](\mathsf{C}[M_1(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}]),\ \ldots,\ \mathsf{C}[M_m(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}])$;[9]

(d) if $A(\boldsymbol{x}) =_* (\forall x : B(\boldsymbol{x}))C(x,\boldsymbol{x})$, where $B(\boldsymbol{x})$ is not a context, then $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}])$ is the set of all Γ_0-acceptable deductions

$$\begin{array}{c}\mathcal{D}\\ M : A(\boldsymbol{A})\end{array}$$

such that if

$$\begin{array}{c}\mathcal{D}'\\ N : B(\boldsymbol{A})\end{array}$$

is in $\mathsf{C}[B(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}])$, then

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ M : A(\boldsymbol{A})\end{array}}{M : (\forall x : B(\boldsymbol{A}))C(x,\boldsymbol{A})}\ (\mathrm{Eq}'') \qquad \begin{array}{c}\mathcal{D}'\\ N : B(\boldsymbol{A})\end{array}}{MN : C(N,\boldsymbol{A}),}\ (\forall\,\mathrm{e})$$

is in $\mathsf{C}[C(N,\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}])$;[10]

(e) if $A(\boldsymbol{x}) =_* (\forall x : B(\boldsymbol{x}))C(x,\boldsymbol{x})$ where $B(\boldsymbol{x})$ is a context, then $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[\boldsymbol{A}])$ is the set of all Γ_0-acceptable deductions

$$\begin{array}{c}\mathcal{D}\\ M : A(\boldsymbol{A})\end{array}$$

such that if

$$\mathcal{D}'$$
$$E : B(A)$$

is in $\mathsf{C}[B(x)](p[A])$ and if $p[E]$ is any computability predicate assigned to E, then

$$\dfrac{\dfrac{\mathcal{D}}{M : A(A)}}{M : (\forall x : B(A))C(x, A)}\ (\mathrm{Eq}'') \qquad \dfrac{\mathcal{D}'}{E : B(A)}}{ME : C(E, A),}\ (\forall\, \mathrm{e})$$

is in $\mathsf{C}[C(x,x)](p[E],p[A])$; and
(f) if $A(x) =_* \lambda x : B(x).C(x,x)$ and is not a Γ_0-proof, then $\mathsf{C}[A(x)](p[A])$ is a function whose argument is a computability function of type A, where A is a Γ_0-term of type $B(A)$ (the type of x), and whose values are given by $(\mathsf{C}[A(x)](p[A]))(p[A])$ $= \mathsf{C}[C(x,x)](p[A],p[A]))$.

Lemma 4.4 (a) *If*

$$x : B$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$$

for $n \geq 0$ is a deduction of type $A(A)$, and if $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are all SN, then

$$x : B$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$$

[9]This definition makes sense only if $\mathsf{C}[A(x)](p[A])$ is a computability predicate. This will be proved below (Lemma 4.6).

[10]In case (d), note that since $B(x)$ is not a context and since $N : B(A), C(N, x)$ must have the same structure (with respect to the construction of types) as $C(x, x)$. The division into cases between (d) and (e) is precisely the distinction between terms which can, after substitution, change the structure of the type in an essential way, and dealing with this possible change is one of the main difficulties of the proof. Furthermore, in cases (d) and (e) of this definition, we are assuming that x does not occur free in A. Since x does not occur in $B(A)$, this is immediate for those A_i which actually occur in $B(A)$, and for those which do not occur in $C(x, A)$, there is clearly no problem. For those A_i which occur in $C(x, A)$ but not in $B(A)$, since we automatically change bound variables to avoid clashes when we carry out a substitution, the fact that the bound variable is x implies that it does not occur free in these A_i.

is in $\mathsf{C}[A(x)](p[A])$.

(b) *Every deduction in* $\mathsf{C}[A(x)](p[A])$ *is* SN.[11]

Proof By induction on the structure of $A(x)$. Note that $A(x)$ is not a Γ_0-proof and does not convert to $\lambda x{:}B(x)\,.\,C(x,x)$.

Case 1. $A(x) =_* \mathsf{Prop}$, Type, or a Γ_0-simple generalized context. Since

$$x : B$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$$

is SN whenever $\mathcal{D}_1$, ..., $\mathcal{D}_n$ are SN, (a) follows by Definition 4.19(b). Part (b) follows immediately by Definition 4.19(b).

Case 2. $A(x) =_* x;M_1 \ldots M_m$ and is not a Γ_0-generalized context. Part (a) holds by Definition 4.15(c) and Definitions 4.18 and 4.19(b). Part (b) holds by Definition 4.15(a) and Definitions 4.18 and 4.19(b).

Case 3. $A(x) =_* (\forall x : B(x))C(x,x)$, where $B(x)$ is not a context. To prove (a), let

$$\mathcal{D}$$
$$M : A(A)$$

be a deduction in $\mathsf{C}[A(x)](p[A])$ and let $x : B(A)$ be an assumption in Γ_0 for which x does not occur free in $\mathcal{D}$. (We may assume without loss of generality that the bound variable x has been changed if necessary to assure that there is such an assumption in Γ_0.) By the induction hypothesis (a) (with $n = 0$), $x : B(A)$ is in $\mathsf{C}[B(x)](p[A])$. Hence, by Definition 4.19(d),

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ M : A(A)\end{array}}{M : (\forall x : B(A))C(x,A)}\,(\mathrm{Eq}'') \qquad x : B(A)}{Mx : C(x,A)}\,(\forall\,\mathrm{e})$$

is in $\mathsf{C}[C(x,x)](p[A])$. Hence, by the induction hypothesis (b), this deduction is SN. Hence, $\mathcal{D}$ is SN.

[11] Cf. Hindley & Seldin [HS86] Theorem A2.3, Lemma 1.

To prove (b), let

$$y : E$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$$

be a Γ_0-acceptable deduction of type A(A) where $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are all SN, and let

$$\mathcal{D}$$
$$N : B(A)$$

be in C$[B(x)](p[A])$. By the induction hypothesis (b), $\mathcal{D}$ is SN. Hence, by the induction hypothesis (a),

$$y : E$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n, \mathcal{D}\}$$

is in C$[C(N, x)](p[A])$. Hence, by Definition 4.19(d),

$$y : E$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$$

is in C$[A(x)](p[A])$.

Case 4. $A(x) =_* (\forall x : B(x))C(x, x)$, where $B(x)$ is a context. To prove (a), let

$$\mathcal{D}$$
$$M : A(A)$$

be in C$[A(x)](p[A])$, and let $x : B(A)$ be an assumption in Γ_0. By the induction hypothesis (a) (with $n = 0$), $x : B(A)$ is in C$[B(x)](p[A])$. By Definition 4.19(e),

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ M : A(A)\end{array}}{M : (\forall x : B(A))C(x, A)}\ (\mathrm{Eq}'') \qquad x : B(A)}{Mx : C(x, A)}\ (\forall\ \mathrm{e})$$

is in C$[C(x, x)](p[x], p[A])$ for all $p[x]$. By the induction hypothesis (b), it is SN. Hence, $\mathcal{D}$ is SN.

To prove (b), let

$$y : E$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$$

be an Γ_0-acceptable deduction of type $A(A)$ where $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are all SN, and let

$$\mathcal{D}$$
$$F : B(A)$$

be in $\mathrm{C}[B(x)](p[A])$. By the induction hypothesis (b), $\mathcal{D}$ is SN. Hence, by the induction hypothesis (a),

$$y : E$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n, \mathcal{D}\}$$

is in $\mathrm{C}[C(x,x)](p[F], p[A])$ for all $p[F]$. Hence, by Definition 4.19(d),

$$y : E$$
$$\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$$

is in $\mathrm{C}[A(x)](p[A])$. ■

Lemma 4.5 *If $\mathcal{D}_1(N)$ is a part of a deduction obtained from a deduction*

$$x : E$$
$$\mathcal{D}_1(x)$$
$$M : B$$

by substituting N for x, if $\mathcal{D}_3$ is SN, *and if*

$$\frac{\begin{array}{c}\mathcal{D}_3 \\ N : C\end{array}}{N : E} \quad (\mathrm{Eq}'')$$

$$\begin{array}{c}\mathcal{D}_1(N) \\ [N/x]M : [N/x]B \\ \{\mathcal{D}_1', \ldots . \mathcal{D}_n'\}\end{array} \qquad (4.6)$$

is in $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[A])$, *then*

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}1\\ [x:E]\\ \mathcal{D}_1(x)\\ M:B\end{array}\qquad\begin{array}{c}\\ \\ \mathcal{D}_2\\ E:\kappa\end{array}}{\lambda x{:}E\,.\,M:(\forall x:E)B}\,(\forall \kappa i-1)}{\lambda x{:}A\,.\,M:(\forall x:C)B}\,(\mathrm{Eq}'')\qquad\begin{array}{c}\\ \\ \mathcal{D}_3\\ N:C\end{array}}{(\lambda x{:}A:M)N:[N/x]B}\,(\forall\,\mathrm{e})$$

$$\{\mathcal{D}_1',\ldots.\mathcal{D}_n'\} \tag{4.7}$$

is also in $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[A])$.[12]

Proof By induction on the structure of $A(\boldsymbol{x})$. Again, $A(\boldsymbol{x})$ is not a Γ_0-proof and does not convert to $\lambda x{:}B(x)\,.\,C(x,\boldsymbol{x})$.

Case 1. $A(\boldsymbol{x}) =_* \mathsf{Prop}$, Type, or a Γ_0-simple generalized context. The lemma follows from Definition 4.19(b) and the fact that (4.7) is SN whenever (4.6) is and the hypotheses of the lemma are satisfied.

Case 2. $A(\boldsymbol{x}) =_* x_1M_1\ldots M_m$ and is not a Γ_0-simple generalized context. The lemma holds by Definition 4.15(b) and Definition 4.19(c).

Case 3. $A(\boldsymbol{x}) =_* (\forall x:B(\boldsymbol{x}))C(x,\boldsymbol{x})$, where $B(\boldsymbol{x})$ is not a context. By hypothesis, (4.6) is in $\mathsf{C}[A(\boldsymbol{x})](\boldsymbol{p}[A])$. Let

$$\begin{array}{c}D\\ P:B(A)\end{array}$$

[12] Cf. Hindley & Seldin [HS86] Theorem A2.3 Lemma 2.

be any deduction in $\mathrm{C}[B(x)](p[A])$. Then by Definition 4.19(d) we have

$$\dfrac{\begin{array}{c}\mathcal{D}_3\\ N : C\end{array}}{\begin{array}{c}N : E\\ \mathcal{D}_1(N)\\ [N/x]M : [N/x]B\{\mathcal{D}_1', \ldots . \mathcal{D}_n', \mathcal{D}\}\end{array}}\ (\mathrm{Eq}'')$$

is in $\mathrm{C}[C(P,x)](p[A])$. By the induction hypothesis,

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}1\\ [x : E]\\ \mathcal{D}_1(x)\\ M : B\end{array}\qquad\begin{array}{c}\\ \\ \mathcal{D}_2\\ E : \kappa\end{array}}{\lambda x{:}E\,.\,M : (\forall x : E)B}\ (\forall \kappa \mathrm{i}-1)}{\lambda x{:}E\,.\,M : (\forall x : C)B}\ (\mathrm{Eq}'')\qquad\begin{array}{c}\mathcal{D}_3\\ N : C\end{array}}{\begin{array}{c}(\lambda x{:}E : M)N : [N/x]B\\ \{\mathcal{D}_1', \ldots . \mathcal{D}_n', \mathcal{D}\}\end{array}}\ (\forall\,\mathrm{e})$$

is in $\mathrm{C}[C(P,x)](p[A])$. Hence, by Definition 4.19(d), (4.7) is in $\mathrm{C}[A(x)](p[A])$.

Case 4. $A(x) =_* (\forall x : B(x))C(x,x)$, where $B(x)$ is a context. By hypothesis, (4.6) is in $\mathrm{C}[A(x)](p[A])$. Let

$$\begin{array}{c}\mathcal{D}\\ F : B(A)\end{array}$$

be any deduction in $\mathrm{C}[B(x)](p[A])$, and let $p[F]$ be a computability function for F.

Then by Definition 4.19(e) we have

$$\dfrac{\begin{array}{c}\mathcal{D}_3\\ N : C\end{array}}{N : E}\ (\mathrm{Eq}'')$$
$$\mathcal{D}_1(N)$$
$$[N/x]M : [N/x]B$$
$$\{\mathcal{D}_1', \ldots . \mathcal{D}_n', \mathcal{D}\}$$

is in $\mathsf{C}[C(x,\boldsymbol{x})](p[F], p[A])$. By the induction hypothesis,

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}1\\ [x : E]\\ \mathcal{D}_1(x)\\ M : B\end{array}}{\lambda x{:}E\ .\ M : (\forall x : E)B}\ (\forall \kappa \mathrm{i} - 1) \qquad \begin{array}{c}\mathcal{D}_2\\ E : \kappa\end{array}}{\lambda x{:}E\ .\ M : (\forall x : C)B}\ (\mathrm{Eq}'') \qquad \begin{array}{c}\mathcal{D}_3\\ N : C\end{array}}{(\lambda x{:}E : M)N : [N/x]B}\ (\forall\, \mathrm{e})$$
$$\{\mathcal{D}_1', \ldots . \mathcal{D}_n', \mathcal{D}\}$$

is in $\mathsf{C}[C(x,\boldsymbol{x})](p[F], p[A])$. Hence, by Definition 4.19(e), (4.7) is in $\mathsf{C}[A(\boldsymbol{x})](p[A])$. ■

Lemma 4.6 *If $A(\boldsymbol{x})$ and $p[A]$ satisfy the hypothesis of Definition 4.19, then $\mathsf{C}[A(\boldsymbol{x})](p[A])$ is a ground for each term $A(A)$.*

Proof Lemmas 4.4 and 4.5. ■

The following lemma makes sense because of Lemma 4.6.

Lemma 4.7 (Substitution) *Let x be a variable which is not assigned a supercontext as a type by Γ_0, let $A(x,y)$ be any Γ_0-type, and let $B(y)$ be a term which can be shown from Γ_0 to have the same type as x, where y includes all variables except*

x which occur free and which are not assigned supercontexts as types by Γ_0. Let $\boldsymbol{C}$ be a sequence of terms of the same types as the variables in $\boldsymbol{y}$ and let $\boldsymbol{p}[\boldsymbol{C}]$ be an assignment of computability predicates to the terms in $\boldsymbol{C}$. Then

$$\mathsf{C}[A(\boldsymbol{x},\boldsymbol{y})](\mathsf{C}[B(\boldsymbol{C})](\boldsymbol{p}[\boldsymbol{C}]),\boldsymbol{p}[\boldsymbol{C}]) = \mathsf{C}[A(B(\boldsymbol{y}),\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{C}]).$$

Proof By induction first on the rank of the type of $B(\boldsymbol{y})$ and second on the structure of $A(\boldsymbol{x},\boldsymbol{y})$. For simplicity, let $p[B(C)]$ abbreviate $\mathsf{C}[B(\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{C}])$. (This is a computability predicate by Lemma 4.6.)

Case 1. $A(\boldsymbol{x},\boldsymbol{y})$ is a Γ_0-proof. Then both sides are $A(B(\boldsymbol{C}),\boldsymbol{C})$ by Definition 4.19(a).

In the remaining cases, we may assume that $A(\boldsymbol{x},\boldsymbol{y})$ is not a Γ_0-proof.

Case 2. x does not occur free in $A(\boldsymbol{x},\boldsymbol{y})$. Then the lemma is trivial. This takes care of the cases in which $A(\boldsymbol{x},\boldsymbol{y})$ converts to Prop or Type.

Case 3. $A(\boldsymbol{x},\boldsymbol{y}) =_* zM_1 \ldots M_n$, a simple generalized context. Then z is assigned a supercontext as a type by Γ_0 and hence, by hypothesis, is distinct from x. Then by Definition 4.19(b), each side consists of the set of all SN deductions of type $A(B(\boldsymbol{C}),\boldsymbol{C})$.

Case 4. $A(\boldsymbol{x},\boldsymbol{y}) =_* yM_1(\boldsymbol{x},\boldsymbol{y})\ldots M_n(\boldsymbol{x},\boldsymbol{y})$, where $y \not\equiv x$ is one of the variables in $\boldsymbol{y}$, and C is the term in $\boldsymbol{C}$ corresponding to y. Then

$$\begin{aligned}&\mathsf{C}[A(\boldsymbol{x},\boldsymbol{y})](p[B(\boldsymbol{C})],\boldsymbol{p}[\boldsymbol{C}]) = \\ &\quad p[C](\mathsf{C}[M_1(\boldsymbol{x},\boldsymbol{y})](p[B(\boldsymbol{C})],\boldsymbol{p}[\boldsymbol{C}]),\ldots,\mathsf{C}[M_n(\boldsymbol{x},\boldsymbol{y})](p[B(\boldsymbol{C})],\boldsymbol{p}[\boldsymbol{C}])),\end{aligned}$$

and since $A(B(\boldsymbol{y}),\boldsymbol{y}) =_* yM_1(B(\boldsymbol{y}),\boldsymbol{y})\ldots M_n(B(\boldsymbol{y}),\boldsymbol{y})$,

$$\mathsf{C}[A(B(\boldsymbol{y}),\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{C}]) = (p[C])(\mathsf{C}[M_1(B(\boldsymbol{y}),\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{C}]),\ldots,\mathsf{C}[M_n(B(\boldsymbol{y}),\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{C}])).$$

The lemma follows by the induction hypothesis.

Case 5. $A(\boldsymbol{x},\boldsymbol{y}) =_* xM_1(\boldsymbol{x},\boldsymbol{y})\ldots M_p(\boldsymbol{x},\boldsymbol{y})$. For simplicity, write this as $x\boldsymbol{M}(\boldsymbol{x},\boldsymbol{y})$. Then the type of x and $B(\boldsymbol{y})$ is

$$(\forall z_1 : E_1)\ldots(\forall z_p : E_p)G,$$

where G is either Prop or a Γ_0-simple context function, and so $B(\boldsymbol{y})$ is a proposition function. By Definition 4.19(c),

$$\mathsf{C}[A(\boldsymbol{x},\boldsymbol{y})](p[B(\boldsymbol{C})],\boldsymbol{p}[\boldsymbol{C}]) = p[B(\boldsymbol{C})](\mathsf{C}[\boldsymbol{M}(\boldsymbol{x},\boldsymbol{y})](p[B(\boldsymbol{C})],\boldsymbol{p}[\boldsymbol{C}])).$$

By the induction hypothesis, the right-hand side equals

$$p[\boldsymbol{C}](\mathsf{C}[\boldsymbol{M}(\boldsymbol{C},\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{C}])),$$

which, by our abbreviation for $p[B(C)]$, is

$$\mathsf{C}[B(y)](p[C])(\mathsf{C}[M(B(y),y)](p[C])).$$

If $p = 0$, we are finished, since $A(B(y),y) =_* B(y)$ and $M(B(y))$ is void, so this is just

$$\mathsf{C}[A(B(y),y)](p[C]),$$

as desired. If $p > 0$, then we have the following subcases according to Corollary 4.13.1:

Subcase 5.1. $B(y) =_* \lambda z_1{:}E_1 \ldots \lambda z_p{:}E_p \,.\, F(z,y)$, where z is the sequence $z_1,\ldots,z_p$. By Definition 4.19(f),

$$\mathsf{C}[B(y)](p[C])(\mathsf{C}[MB(y),y)](p[C]))$$

is

$$\mathsf{C}[B(y)z](p[C],\mathsf{C}[MB(y),y)](p[C])).$$

By the induction hypothesis on the type of $B(y)$, this is

$$\mathsf{C}[B(y)M(B(y),y)](p[C]),$$

and since $A(B(y),y) =_* B(y)M(B(y),y)$, we are done.

Subcase 5.2. $B(y) =_* y_i N_1(y) \ldots N_q(y)$, which we may as well abbreviate as $y_i N(y)$. Then $A(B(y),y) =_* y_i N(y) M(B(y),y)$. Now by Definition 4.19(c),

$$\mathsf{C}[B(y)](p[C])(\mathsf{C}[MB(y),y)](p[C]))$$

is

$$p[C_i](\mathsf{C}[N(y)](p[C]))(\mathsf{C}[MB(y),y)](p[C])),$$

but this is the same thing as

$$p[C_i](\mathsf{C}[N(y)](p[C]),\mathsf{C}[MB(y),y)](p[C])),$$

and by Definition 4.19(c), this is

$$\mathsf{C}[A(B(y),y)](p[C]),$$

as desired.

Case 6. $A(x,y) =_* (\forall z : E(x,y))F(z,x,y)$, where $E(x,y)$ is not a context. By the induction hypothesis,

$$\mathsf{C}[E(x,y)](p[B(C)],p[C]) = \mathsf{C}[E(B(y),y)](p[C])$$

and, for any term $N(y)$ such that there is a Γ_0-acceptable deduction ending in $N(C) : E(B(C))$,

$$\mathsf{C}[F(z,x,y)](p[B(C)],p[C]) = \mathsf{C}[F(z,B(y),y)](p[C]).$$

By Definition 4.19(d), the lemma follows.

Case 7. $A(x,y) =_* (\forall z : E(x,y))F(z,x,y)$, where $E(x,y)$ is a context. Similar to Case 4 using Definition 4.19(e). ■

Notation In the following lemma, x will denote the sequence $x_1,\ldots,x_n$, y the sequence $y_1,\ldots,y_m$, N the sequence $N_1,\ldots,N_n$, B the sequence $B_1,\ldots,B_m$, and $p[B]$ the sequence $p[B_1],\ldots,p[B_m]$. Furthermore, A'_{i+1}, for $i = 0,1,\ldots,n-1$, will denote $[N_1/x_1,\ldots,N_i/x_i]A_{i+1}$.

Lemma 4.8 *Let*

$$x_1 : A_1(y),\ldots,x_n : A_n(y)$$
$$\mathcal{D}(x,y)$$
$$M(x,y) : A(x,y)$$

be a Γ_0-acceptable deduction all of whose undischarged assumptions are among those shown, where y consists of all variables which occur free in any type or term which are not assigned supercontexts as types by Γ_0. For all assignments of terms $B_1,\ldots,B_m$ to $y_1,\ldots,y_m$ (where for each $i = 1,2,\ldots,m$, it can be proved from Γ_0 that B_i is in the type assigned to y_i) and for all assignments of computability predicates $p[B_1]$, ..., $p[B_m]$ to $B_1,\ldots,B_m$, if for $i = 1,2,\ldots,n$, the Γ_0-acceptable deduction

$$\mathcal{D}_i$$
$$N_i : A'_i(B)$$

is in $\mathsf{C}[A_i(y)](p[B])$, then

$$\begin{array}{c} \mathcal{D}_1 \qquad\qquad \mathcal{D}_n \\ N_1 : A'_1(B)\ ,\ldots,\ N_n : A'_n(B) \\ \mathcal{D}(N,B) \\ M(N,B) : A(N,B), \end{array} \tag{4.8}$$

is in $\mathsf{C}[A(N,y)](p[B])$.[13]

Proof By induction on structure of $\mathcal{D}(x,y)$.

Basis:

Case 1. $\mathcal{D}(x,y)$ consists of the axiom (P T). Since this deduction is clearly SN, the lemma follows by Definition 4.19(b).

Case 2. $\mathcal{D}(x,y)$ consists of the assumption $x_i : A_i(y)$. The lemma is immediate.

Induction step: There are the following cases, according to the last inference in $\mathcal{D}(x,y)$.

Case 1. The last inference is by ($\kappa\kappa'$Formation). By Definition 4.19(b), it is sufficient to prove that (4.8) is SN. By the induction hypothesis and Definition 4.19(b), the deductions of both premises are SN. Hence, (4.8) is SN.

Case 2. The last inference is by (Eq'κ). Similar to Case 1.

Case 3. The last inference is by ($\forall$ e). Then $M(x,y) \equiv M_1(x,y)M_2(x,y)$, $A(x,y) \equiv E(M_2(x,y),x,y)$,
and $\mathcal{D}(x,y)$ is

$$\frac{\begin{array}{cc} x_1 : A_1(y),\ldots,x_n : A_n(y) & x_1 : A_1(y),\ldots,x_n : A_n(y) \\ \mathcal{D}'(x,y) & \mathcal{D}''(x,y) \\ M_1(x,y) : (\forall x : C(x,y))E(x,x,y) & M_2(x,y) : C(x,y) \end{array}}{M_1(x,y)M_2(x,y) : E(M_2(x,y),x,y).} \quad (\forall \text{ e})$$

Subcase 3.1. $C(x,y)$ is not a context. By the induction hypothesis,

$$\begin{array}{c} \mathcal{D}_1 \qquad\qquad \mathcal{D}_n \\ N_1 : A_1'(B) \ ,\ldots,\ N_n : A_n'(B) \\ \mathcal{D}'(N,B) \\ M_1(N,B) : (\forall x : C(N,B))E(x,N,B) \end{array}$$

is in $\mathsf{C}[(\forall x : C(N,y))E(x,N,y)](p[B])$ and

$$\begin{array}{c} \mathcal{D}_1 \qquad\qquad \mathcal{D}_n \\ N_1 : A_1'(B) \ ,\ldots,\ N_n : A_n'(B) \\ \mathcal{D}''(N,B) \\ M_2(N,B) : (N,B), \end{array}$$

[13] Cf. Hindley & Seldin [HS86] Theorem A2.3 Lemma 3(b).

is in $\mathsf{C}[C(\boldsymbol{N},\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{B}])$. Then by Definition 4.19(d), (4.8) is in $\mathsf{C}[E(M2(\boldsymbol{N},\boldsymbol{y}),\boldsymbol{N},\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{B}])$.

Subcase 3.2. $C(\boldsymbol{x},\boldsymbol{y})$ is a context. By the induction hypothesis,

$$\begin{array}{c} \mathcal{D}_1 \qquad\qquad \mathcal{D}_n \\ N_1 : A_1'(B)\ ,\ldots,\ N_n : A_n'(B) \\ \mathcal{D}'(\boldsymbol{N},\boldsymbol{B}) \\ M_1(\boldsymbol{N},\boldsymbol{B}) : (\forall x : C(\boldsymbol{N},\boldsymbol{B}))E(x,\boldsymbol{N},\boldsymbol{B}) \end{array}$$

is in $\mathsf{C}[(\forall x : C(\boldsymbol{N},\boldsymbol{y}))E(x,\boldsymbol{N},\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{B}])$ and

$$\begin{array}{c} \mathcal{D}_1 \qquad\qquad \mathcal{D}_n \\ N_1 : A_1'(B)\ ,\ldots,\ N_n : A_n'(B) \\ \mathcal{D}(\boldsymbol{N},\boldsymbol{B}) \\ M(\boldsymbol{N},\boldsymbol{B}) : A(\boldsymbol{N},\boldsymbol{B}), \end{array}$$

is in $\mathsf{C}[C(\boldsymbol{N},\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{B}])$. Then by Definition 4.19(e), for any computability predicate $p[M_2(\boldsymbol{N},\boldsymbol{B})]$, (4.8) is in $\mathsf{C}[E(x,\boldsymbol{N},\boldsymbol{y})](p[M_2(\boldsymbol{N},\boldsymbol{B})],\boldsymbol{p}[\boldsymbol{B}])$. To complete the proof, it is sufficient to find a computability predicate $p[M_2(\boldsymbol{N},\boldsymbol{y})]$ such that

$$\mathsf{C}[E(x,\boldsymbol{N},\boldsymbol{y})](p[M_2(\boldsymbol{N},\boldsymbol{B})],\boldsymbol{p}[\boldsymbol{B}]) = \mathsf{C}[E(M_2(\boldsymbol{N},\boldsymbol{y}),\boldsymbol{N},\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{B}]). \qquad (4.9)$$

A suitable such function is the one such that

$$p[M_2(\boldsymbol{N},\boldsymbol{B})] = \mathsf{C}[M_2(\boldsymbol{N},\boldsymbol{y})](\boldsymbol{p}[\boldsymbol{B}]).$$

That this is a computability predicate follows from Definition 4.18 and Lemma 4.6. That (4.9) holds follows from Lemma 4.7.

Case 4. The last inference is by $(\forall\kappa\mathrm{i})$. Then $A(\boldsymbol{x},\boldsymbol{y}) \equiv (\forall x : C(\boldsymbol{x},\boldsymbol{y}))E(x,\boldsymbol{x},\boldsymbol{y})$, $M(\boldsymbol{x},\boldsymbol{y})$ is $\lambda x{:}C(\boldsymbol{x},\boldsymbol{y})\ .\ M_1(x,\boldsymbol{x},\boldsymbol{y})$, and $\mathcal{D}(\boldsymbol{x},\boldsymbol{y})$ is

$$\dfrac{\begin{array}{cc} 1 & \\ [x : C(\boldsymbol{x},\boldsymbol{y})],\ x_1 : A_1(\boldsymbol{y}),\ldots,x_n : A_n(\boldsymbol{y}) & x_1 : A_1(\boldsymbol{y}),\ldots,x_n : A_n(\boldsymbol{y}) \\ \mathcal{D}'(x,\boldsymbol{x},\boldsymbol{y}) & \mathcal{D}''(\boldsymbol{x},\boldsymbol{y}) \\ M_1(x,\boldsymbol{x},\boldsymbol{y}) : E(x,\boldsymbol{x},\boldsymbol{y}) & C(\boldsymbol{x},\boldsymbol{y}) : \kappa \end{array}}{\lambda x{:}C(\boldsymbol{x},\boldsymbol{y})\ .\ M_1(x,\boldsymbol{x},\boldsymbol{y}) : (\forall x : C(\boldsymbol{x},\boldsymbol{y}))E(x,\boldsymbol{x},\boldsymbol{y})}\ (\forall\kappa\mathrm{i}-1)$$

Subcase 4.1. $C(x,y)$ is not a context. Then $\kappa \equiv \mathsf{Prop}$. By the induction hypothesis, for all deductions

$$\mathcal{D}'''$$
$$P : C(N,B)$$

in $\mathsf{C}[C(N,y)](p[B])$,

$$\begin{array}{c} \mathcal{D}''' \qquad \mathcal{D}_1 \qquad\quad \mathcal{D}_n \\ P : C(N,B)\ ,\ N_1 : A_1'(B)\ \ldots\ N_n : A_n'(B) \\ \mathcal{D}'(P,N,B) \\ M_1(P,N,B) : E(P,N,B) \end{array}$$

is in $\mathsf{C}[E(P,N,y)](p[B])$. Hence, by Lemmas 4.4(b) and 4.5,

$$\dfrac{\dfrac{\begin{array}{c} 1 \qquad \mathcal{D}_1 \qquad\quad \mathcal{D}_n \qquad\quad \mathcal{D}_1 \qquad\quad \mathcal{D}_n \\ [x : C^*],\ N_1 : A_1^*,\ldots,N_n : A_n^* \qquad N_1 : A_1^*,\ldots,N_n : A_n^* \\ \mathcal{D}'^*(x) \qquad\qquad\quad \mathcal{D}''^* \\ M_1^*(x) : E^*(x) \qquad\qquad C^* : \kappa \end{array}}{\lambda x{:}C^*\ .\ M_1^*(x) : (\forall x : C^*)E^*(x)}\ (\forall \kappa \mathrm{i}-1) \qquad \begin{array}{c}\mathcal{D}''' \\ P : C^*\end{array}}{(\lambda x{:}:C^*)\ .\ M_1^*(x)P : E^*(P),}\ (\forall\,\mathrm{e})$$

where $A_i^* \equiv A_i'(B)$, $X^* \equiv X(N,B)$, and $X^*(Y) \equiv X(Y,N,B)$, is also in $\mathsf{C}[E(P,N,y)](p[B])$. Since $\mathcal{D}'''$ is arbitrary, this implies by Definition 4.19(e) that (4.8) is in $\mathsf{C}[A(N,y)](p[B])$.

Subcase 4.2. $C(x,y)$ is a context. Then $\kappa \equiv \mathsf{Type}$. By the induction hypothesis, for all deductions

$$\mathcal{D}'''$$
$$F : C(N,B)$$

in $\mathsf{C}[C(N,y)](p[B])$ and for all computability predicates $p[F]$,

$$\begin{array}{c} \mathcal{D}''' \qquad \mathcal{D}_1 \qquad\quad \mathcal{D}_n \\ F : C(N,B)\ ,\ N_1 : A_1'(B)\ \ldots\ N_n : A_n'(B) \\ \mathcal{D}'(F,N,B) \\ M_1(F,N,B) : E(F,N,B) \end{array}$$

is in $\mathsf{C}[E(x,N,y)](p[F],p[B])$. Hence, by Lemmas 4.4(b) and 4.5,

$$\dfrac{\dfrac{\begin{array}{cccc} 1 & \mathcal{D}_1 & \mathcal{D}_n & \mathcal{D}_1 \quad \mathcal{D}_n \\ [x:C^*], & N_1:A_1^*,\ldots,N_n:A_n^* & & N_1:A_1^*,\ldots,N_n:A_n^* \\ & \mathcal{D}'^*(x) & & \mathcal{D}''^* \\ & M_1^*(x):E^*(x) & & C^*:\kappa \end{array}}{\lambda x{:}C^*\,.\,M_1^*(x):(\forall x:C^*)E^*(x)}\,(\forall\kappa\mathrm{i}-1) \quad \begin{array}{c}\mathcal{D}''' \\ F:C^*\end{array}}{(\lambda x::C^*)\,.\,M_1^*(x)F:E^*(F),}\,(\forall\,\mathrm{e})$$

where A_i^*, X^*, and $X^*(Y)$ are as in Subcase 4.1, is also in $\mathsf{C}[E(x,N,y)](p[F],p[B])$. Since $\mathcal{D}'''$ and $p[F]$ are arbitrary, this implies by Definition 4.19(d) that (4.8) is in $\mathsf{C}[A(N,y)](p[B])$.

Case 5. The last inference is by (Eq''). This is straightforward by Definition 4.19.

Case 6. The last inference is by ($\equiv'_\alpha$). This is trivial by Definition 4.19. ■

Theorem 4.14 (Strong normalization) *Every deduction in TAC is strongly normal.*

Proof In Lemma 4.8, let $\mathcal{D}_i$ consist of the assumption $x_i:A_i(y)$ and let B_j be y_j. Then for any sequence $p[B]$, $\mathcal{D}(x,y)$ is in $\mathsf{C}[A(x,y)](p[B])$, and so is SN. ■

4.4 Consequences of the strong normalization theorem

Although we have proved the strong normalization theorem for deductions, this theorem is usually proved for terms. We saw in Theorem 2.2 and Corollary 2.2.1 that for TA, the normalization theorem for terms can be proved from the strong normalization theorem for deductions by using the subject-construction theorem. We do not have this theorem for TAC in a form that is easy to state. Nevertheless, there is a relationship between terms and deductions, and we can expect to use this relationship to obtain a normalization theorem for terms.

Theorem 4.15 (Normalization theorem for terms) *If* Γ *is a well-formed environment and if*

$$\Gamma \vdash_{\mathrm{TAC}} M : A,$$

then M *has a normal form.*

Proof By Theorem 4.14 there is a normal deduction $\mathcal{D}$ of

$$\Gamma \vdash_{\mathrm{TAC}} N : A,$$

where $M N$. The proof is by induction on the deduction $\mathcal{D}$.

Basis: If $\mathcal{D}$ consists of an assumption, then N is a variable, and so it is in normal form. If $\mathcal{D}$ consists of the axiom $(\mathsf{P}\ \mathsf{T})$, then N is Prop, which is in normal form.

Induction step: There are the following cases, depending on the last inference in $\mathcal{D}$.

Case 1. The last inference is by rule ($\kappa\kappa'$Formation). Then A is κ', N is $(\forall x : B)C$, and $\mathcal{D}$ is

$$\dfrac{\begin{array}{ccc} & 1 & \\ & [x : B] & \\ \mathcal{D}_1 & \mathcal{D}_2(x) & \\ b : \kappa & C : \kappa' & \end{array}}{(\forall x : B)C : \kappa'}\ \kappa\kappa'\text{Formation}$$

By the induction hypothesis, B and C have normal forms; hence, so does A.

Case 2. The last inference is by rule (Eq'κ). Then by the induction hypothesis, N converts to a term B (to the left of the colon in the premise) which has a normal form.

Case 3. The last inference is by rule (∀e). Then $N \equiv PQ$, $A \equiv [Q/x]C$, and $\mathcal{D}$ is

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ P : (\forall x : B)C & Q : B\end{array}}{PQ : [Q/x]C.}\ (\forall\,\mathrm{e})$$

By the induction hypothesis, P and Q have normal forms. Furthermore, since $\mathcal{D}$ is normal, there is no κ-reduction possible in it. It follows that at the top of the left branch of $\mathcal{D}$ (and hence of $\mathcal{D}_1$) is an undischarged assumption. It follows that $P =_* yQ_1 \ldots Q_m$ for some variable y. It follows that $Q_1, \ldots, Q_m$ all have normal forms, and hence that $PQ =_* yQ_1 \ldots Q_m Q$ does as well.

Case 4. The last inference is by rule (∀κi). Then $A \equiv (\forall x : B)C$, $N \equiv \lambda x{:}B\,.\,P$, and $\mathcal{D}$ is

$$\dfrac{\begin{array}{cc}1 & \\ [x : B] & \\ \mathcal{D}_1(x) & \mathcal{D}_2 \\ P : C & B : \kappa\end{array}}{\lambda x{:}B\,.\,P\,.\,(\forall x : B)C.}\ (\forall\kappa\mathrm{i} - 1)$$

By the induction hypothesis, B and P have normal forms; hence, so does $N \equiv \lambda x{:}B\,.\,P$.

Case 5. The last inference is by rule (Eq″). Then N is the term to the left of the colon in the premise, and so by the induction hypothesis it has a normal form.

Case 6. The last inference is by rule $(\equiv'_\alpha)$. Then N is obtained by changes of bound variables from a term which, by the induction hypothesis, has a normal form, and so N has a normal form. ■

Note that we have not proved that every term is SN. If we try to replace the conclusion by "N is SN" in the above proof, we can see that Case 2 breaks down, since not every term convertible to an SN term is itself SN. Indeed, if A is SN, and if $x \notin \mathrm{FV}(A)$, then for any terms B and C, $(\lambda x{:}B\,.\,A)C =_* A$; now if C has no normal form, then $(\lambda x{:}B\,.\,A)C$ is not SN. This shows that we cannot strengthen the theorem to prove that N is SN. (Of course, to prove that M is SN is somewhat more complicated; we will take this up below.)

It might appear that since only Case 2 breaks down, and since the conclusion in this case is not a proof, we might want to add the assumption that $N : A$ is a

proof. This will exclude Case 2. But now we have trouble with Case 4: we can conclude that P is SN, but not that B is SN. Indeed, by the remarks of the previous paragraph, B might not be SN.

Mitchell [Mit86] defines a function *Erase* for TAP which deletes the types of the bound variables. When this function is modified for TAC, it is defined as follows:

Definition 4.20 (Erase function)
(a) $Erase(a) \equiv a$ if a is a constant or a variable;
(b) $Erase(MN) \equiv Erase(M)Erase(N)$;
(c) $Erase(\lambda x{:}A\,.\,M) \equiv \lambda x\,.\,Erase(M)$; and
(d) $Erase((\forall x : A)B) \equiv (\forall x : Erase(A))Erase(B)$.

Note that except for clause (d), we are mapping terms of TAC to pure λ-terms. In fact, the range of the function *Erase* is the set of TAG terms (Definition 2.17).

We can now prove that if A is not a context in the theorem, then $Erase(N)$ is SN. To extend this result to $Erase(M)$, it is enough to note that deductions of proofs do follow the constructions of the terms except that additional inferences of formulas which are not proofs are added at various places on top. This will give us the following result:

Corollary 4.15.1 *Under the hypotheses of Theorem 4.15, if A is not a context, then $Erase(M)$ is strongly normal.*

There are some further corollaries that follow immediately from Theorem 4.15. These corollaries are standard consequences of normalization theorems.

Corollary 4.15.2 *For terms M and N such that*

$$\Gamma \vdash_{\mathrm{TAC}} M : A,$$

and

$$\Gamma \vdash_{\mathrm{TAC}} N : A,$$

where Γ is a well-formed environment, it is decidable whether or not $M =_* N$.

Corollary 4.15.3 *For a terms M and a well-formed environment Γ, it is decidable whether or not there is a term A such that*

$$\Gamma \vdash_{\mathrm{TAC}} M : A.$$

We can also prove a partial converse to Theorem 4.2, relating TAC to TAP. Recall[14] that the interpretation of types and terms of TAP as terms of TAC is defined as follows: first, we divide the variables of TAC into two mutually disjoint classes, the first for interpreting term variables of TAP and the second for interpreting the type variables. Then, for a term or type A of TAP, we define A^*, a term of TAC, as follows:

(a) if x is a term variable, then x^* is a variable of the first class distinct from all variables y^* for term variables y distinct from x;

(b) if a is a type variable, then a^* is a variable of the second class distinct from all variables b^* for type variables b distinct from a;

(b) $(\alpha \to \beta)^*$ is $(\forall x : \alpha)^*\beta^*$ for a (term-) variable x which does not occur free in α^* or
β^*;

(c) $((\forall a)\alpha)^*$ is $(\forall a^* : \mathsf{Prop})\alpha^*$;

(d) $(MN)^*$ is M^*N^*;

(e) $(M\alpha)^*$ is $M^*\alpha^*$;

(f) $\lambda x{:}\alpha \,.\, M^*$ is $\lambda x^* : \alpha^* \,.\, M^*$; and

(g) $\lambda a.M^*$ is $\lambda a^* : \mathsf{Prop} \,.\, M^*$.

It is easy to show that if α is any type-scheme of TAP, then α^* is in normal form, and that if M is any term of TAP which is in normal form, then M^* is also in normal form. Note also that this interpretation takes any β^2-contraction of TAP into a β-contraction of TAC.

Theorem 4.16 *Let Γ be a sequence*

$$x_1 : \alpha_1,\ x_2 : \alpha_2,\ \ldots,\ x_n : \alpha_n$$

of assumptions in TAP, and let Γ^* be

$$x_1^* : \alpha_1^*,\ x_2^* : \alpha_2^*,\ \ldots,\ x_n^* : \alpha_n^*$$

Let α be any type scheme in TAP, let $a_1, \ldots, a_m$ include all of the type variables which occur free in α, and let Γ' be

$$a_1' : \mathsf{Prop}, \ldots, a_m^* : \mathsf{Prop}.$$

If $\mathcal{D}$ is a normal deduction in TAC of

$$\Gamma^*, \Gamma' \vdash M^* : \alpha^*,$$

[14] Cf. Hindley & Seldin [HS86] Theorem 16.66

where M is a term of TAP, then there is a normal deduction $\mathcal{D}'$ in TAP of

$$\Gamma \vdash M : \alpha.$$

Proof Note first that Lemmas 16.67 and 16.68 of Hindley & Seldin [HS86] hold for TAC as well as for TAGL; the proofs for TAC are obtained by a minor change in notation from those for TAGL.

The proof is by induction on the deduction $\mathcal{D}$. Note that by hypothesis, $\mathcal{D}$ does not consist of axiom (P T), and its last inference is not by any of rules ($\kappa\kappa'$Formation) or (Eq$'\kappa$). Furthermore, since we are assuming that $\mathcal{D}$ has been transformed according to Theorem 4.1, we may assume that the last inference is not by rule (Eq$''$). For the types of the assumptions (both discharged and undischarged) are all in normal form, and if the types of the premises of any rule except ($\forall$e) and (Eq$''$) are in normal form, then so is the type of the conclusion. With regard to inferences in $\mathcal{D}$ by rule ($\forall$e) the left branch above each such inference contains inferences only by the same rule and rule (Eq$''$) and at the top of the branch is an assumption (since $\mathcal{D}$ is normal); and it is not hard to see by beginning with the assumption that because the type of the left premise of each such inference by rule ($\forall$e) is β^* for some TAP type scheme β, so is the type of the conclusion. It follows that each of these types is in normal form, and so there is no inference by rule (Eq$''$) in the branch. There are the following remaining cases:

Case 1. $\mathcal{D}$ consists of an assumption. Then M is x_i, α is α_i, and $\mathcal{D}'$ consists of the corresponding assumption in TAP.

Case 2. The last inference in $\mathcal{D}$ is by rule ($\forall$e). Then since $\mathcal{D}$ is normal, the only inferences which occur in the left branch are by rules ($\forall$e). Furthermore, M^* is in normal form. Now it follows from this that M^* has the form $xM_1 \ldots M_p$, where x is assigned a type by the assumption at the top of the branch (which is not discharged). Hence, x is one of the x_i^*. By the definition of the interpretation, it follows that each M_j is either N_j^* for some TAP term N_j, in which case the type assigned to it is γ_j^* for some TAP type scheme γ_j, or else some β_j^* for some TAP type scheme β_j, in which case the type assigned to it is Prop. By the induction hypothesis, there is a normal deduction $\mathcal{D}_j$ of $\Gamma \vdash N_j : \gamma_j$ for each such N_j, and then rules ($\rightarrow$ e) and ($\forall$e) of TAP can be used to obtain $\mathcal{D}'$ from the assumption x_i : α_i and the deductions $\mathcal{D}_j$.

Case 3. The last inference in $\mathcal{D}$ is by rule ($\forall$Pi). Then α^* is $(\forall x : B)C$ and M^* is $\lambda x{:}B\,.\,N$. By the right premise, B is β^* for some TAP type scheme β, and it follows that x is some y^*, for a TAP term variable y, and does not occur free in C; furthermore, C is γ^* for some TAP type scheme γ. In addition, N is P^* for some TAP term P. It followsthat if the last inference is removed from $\mathcal{D}$, the result is a

normal deduction $\mathcal{D}_1$ of

$$\Gamma^*, y^* : \beta^*, \Gamma' \vdash_{\mathrm{TAC}} P^* : \gamma^*.$$

By the induction hypothesis, there is a normal deduction $\mathcal{D}_1'$ of

$$\Gamma, y : \beta \vdash_{\mathrm{TAP}} P : \gamma,$$

and $\mathcal{D}'$ is obtained by an inference by rule $(\rightarrow \mathrm{i})$.

Case 4. The last inference in $\mathcal{D}$ is by rule $(\forall \mathrm{T}i)$. Then α^* is $(\forall x : B)C$ and M^* is $\lambda x : B \,.\, N$. By the right premise, B is Prop. Hence, x is β^* for a TAP type variable a, C is β^* for some TAP type scheme β, and N is P^* for some TAP term P. It follows that if the last inference is removed from $\mathcal{D}$, the result is a normal deduction $\mathcal{D}_1$ of

$$\Gamma^*, \Gamma', a^* : \mathsf{Prop} \vdash_{\mathrm{TAC}} P^* : \beta^*.$$

By the induction hypothesis, there is a normal deduction $\mathcal{D}_1'$ of

$$\Gamma \vdash_{\mathrm{TAP}} P : \beta.$$

Since α is $(\forall a)\beta$, $\mathcal{D}'$ follows by an inference by rule $(\forall \mathrm{i})$.

Case 5. The last inference in $\mathcal{D}$ is by rule $(\equiv'_\alpha)$. This case is trivial since the same rule (essentially) is also a rule of TAP. ■

Corollary 4.16.1 *Under the hypotheses of the theorem, if* $N =_* M^*$ *and if* $A =_* \alpha^*$, *and if*

$$\Gamma^*, \Gamma' \vdash_{\mathrm{TAC}} N : A,$$

then

$$\Gamma \vdash_{\mathrm{TAP}} M : \alpha.$$

4.5 The theory of constructions: sequent formulation

In this section we shall consider an alternative formulation of the theory of constructions. It is a variant of the form in which the theory was originally presented in Coquand [Coq85], and is closer to the presentation in other papers by Coquand and Huet than is the system TAC.

As we saw in the last section, every rule which discharges an assumption of the form $x : A$ has a premise not depending on this discharged assumption that is either A : Prop or A : Type. If we wanted to, we could take these premises as justifications for the assumptions instead of premises for the rules; this is the approach adopted by Martin-Löf in his work (see his [Mar75], [Mar82], and [Mar84]). The main reason this is not done in TAC is that it would require that premise to be written above the assumption, and then the assumptions would not occur at the tops of branches, an inconvenience for the theory of a system such as TAC. But for the form of the theory of constructions presented by Coquand, it is the most useful approach.

This form of the theory of constructions is what is known as a sequent calculus. A sequent is an expression of the form

$$\Gamma \vdash E, \tag{4.10}$$

where Γ is a (possibly empty) sequence of formulas and E is a formula. This particular sequent calculus is formulated in such a way that the only nonempty sequences that can occur to the left of the turnstile (the symbol '$\vdash$') are well-formed environments. This will make unnecessary the premises which "justify" the discharged assumptions; for these assumptions will all occur to the left of the turnstile in the premises of the rules and will hence be part of well-formed environments, and so these premises will automatically hold. The fact that Γ is a well-formed environment will be equivalent to the derivability of the sequent

$$\Gamma \vdash \mathsf{Prop} : \mathsf{Type}.$$

The system will be called TACS.

Note that until the equivalence of TAC and TACS is proved, it will be necessary to specify the system with respect to which an environment is well-formed. Until notice to the contrary is given, a well-formed environment will mean with respect to TACS.

Definition 4.21 (The type assignment system TACS) The system TACS is a sequent calculus; its sequents are of the form

$$\Gamma \vdash E, \tag{4.11}$$

where Γ is a sequence of TAC formulas and E is a TAC formula. The system has one axiom:

$$(\mathsf{P\ T}) \qquad \vdash \mathsf{Prop} : \mathsf{Type}$$

Its rules are as follows, where, in each case, x is a variable which does not occur free in Γ or in A, and κ is any kind:

I. Well-formed environments:

$$(\mathsf{Pi}) \qquad \frac{\Gamma \vdash A : \kappa}{\Gamma, x : A \vdash \mathsf{Prop} : \mathsf{Type}}$$

II. Introduction of product:

$$(\forall \mathsf{i}) \qquad \frac{\Gamma, x : A \vdash B : \kappa}{\Gamma \vdash (\forall x : A)B : \kappa,}$$

III. Introduction of a variable:

$$(\mathsf{Pe}) \qquad \frac{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}}{\Gamma \vdash y : A,}$$

Condition: $y : A$ occurs in Γ and y does not occur free in A.

IV. Lambda introduction:

$$(\lambda \mathsf{i}) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A \,.\, M : (\forall x : A)B,}$$

V. Application:

$$(\forall \mathsf{e}) \qquad \frac{\Gamma \vdash M : (\forall x : A)B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B,}$$

VI. Equality rules:
(Eq″) *If* $A =_* B$, *then*

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash M : A,}$$

(Eq′κ) *If* $A =_* B$, *then*

$$\frac{\Gamma \vdash B : \kappa}{\Gamma \vdash A : \kappa}$$

VII. Changes of bound variables:
If N is obtained from M by changes of bound variables, then:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash N : A.}$$

We shall now establish the equivalence of TACS and TAC:

Lemma 4.9 *If* $\Gamma \vdash_{\text{TACS}} E$ *for any formula* E, *and if* Γ' *is any initial segment of* Γ (*possibly including* Γ *itself*), *then each derivation of* $\Gamma \vdash_{\text{TACS}} E$ *contains a subderivation of* $\Gamma' \vdash_{\text{TACS}}$ Prop : Type.

Proof By induction on the derivation of $\Gamma \vdash_{\text{TACS}} E$.

Basis: If $\Gamma \vdash_{\text{TACS}} E$ is the axiom (P T), then Γ' is empty, and the result is trivial.

Induction step: We assume the property for each premise of a rule and prove it for the conclusion.

If the sequence to the left of $\vdash$ in the conclusion is an initial segment of that of at least one premise, this is trivial. This takes care of all rules except (Pi). In this case, Γ is Γ_1, A : Prop, and E is Prop : Type. If Γ' is all of Γ, then the entire deduction is what we seek. Otherwise, Γ' is an initial segment of Γ_1, and the result is trivial by the induction hypothesis. ■

Lemma 4.10 *If* $\Gamma \vdash_{\text{TACS}}$ Prop : Type, *then* Γ *is a well-formed environment.*

Proof By induction on the pair $\langle n, m \rangle$, where n is the number of formulas in Γ and m is the length of the derivation of $\Gamma \vdash_{\text{TACS}}$ Prop : Type.

Basis: Trivial, since Γ is empty.

Induction step: Assume the lemma for any initial subsequence of Γ, and suppose that Γ is $\Gamma', x : A$. By the induction hypothesis, Γ' is a well-formed environment. Now the only rules of which

$$\Gamma', x : A \vdash_{\mathrm{TACS}} \mathsf{Prop} : \mathsf{Type}$$

can be the conclusion are the equality rules and (Pi). If the rule is an equality rule, then by Lemma 4.9 there is a subderivation of the derivation of the premise of the inference which is a derivation of

$$\Gamma', x : A \vdash_{\mathrm{TACS}} \mathsf{Prop} : \mathsf{Type}$$

and so the conclusion follows by the induction hypothesis; if the rule is (Pi), then it follows that x does not occur free in Γ' or in A and that

$$\Gamma' \vdash_{\mathrm{TACS}} A : \kappa.$$

Since Γ' is a well formed environment, this implies that Γ is as well. ■

Lemma 4.11 *If* $\Gamma \vdash_{\mathrm{TACS}} E$, *then* Γ *is a well-formed environment.*

Proof Lemmas 4.9 and 4.10. ■

Theorem 4.17 *There is a formula* E *such that* $\Gamma \vdash_{\mathrm{TACS}} E$ *if and only if* Γ *is a well-formed environment.*

Proof The "only if" part is Lemma 4.11. The "if" part is easy using the axiom and rules (Pi) . ■

We are now in a position to prove the equivalence between TAC and TACS.

Theorem 4.18 *If*

$$\Gamma \vdash_{\mathrm{TACS}} E, \tag{4.12}$$

then

$$\Gamma \vdash_{\mathrm{TAC}} E. \tag{4.13}$$

Proof By induction on the derivation of (4.12).

Basis: (4.12) is axiom (P T). Then Γ is empty, E is Prop : Type, and (4.13) holds by axiom (P T) in TAC.

Induction step: The cases are by the last rule used in the derivation of (4.12).

Case (Pi). Trivial.

Case ($\forall$i). E is $(\forall x : A)B : \kappa$, where x does not occur free in A or Γ, and the premise is

$$\Gamma, x : A \vdash_{\mathrm{TACS}} B : \kappa.$$

By the induction hypothesis,

$$\Gamma, x : A \vdash_{\mathrm{TAC}} B : K.$$

Furthermore, by Theorem 4.17, $\Gamma, x : A$ is a well-formed environment (with respect to TACS). This means that the derivation of (4.12) includes a subderivation of

$$\Gamma \vdash_{\mathrm{TACS}} A : \kappa'.$$

Hence, again by the induction hypothesis,

$$\Gamma \vdash_{\mathrm{TAC}} A : \kappa'.$$

Hence, (4.13) follows by ($\kappa\kappa'$Formation).

Case (Pe). Trivial by the conventions of natural deduction systems.

Case (λi). Similar to Case ($\forall$i), using ($\forall\kappa$i).

Case ($\forall$e). E is $MN : [N/x]B$, and the premises are

$$\Gamma \vdash_{\mathrm{TACS}} M : C \quad \text{and} \quad \Gamma \vdash_{\mathrm{TACS}} N : A,$$

where $C =_* (\forall x : A)B$. By the induction hypothesis

$$\Gamma \vdash_{\mathrm{TAC}} M : C \quad \text{and} \quad \Gamma \vdash_{\mathrm{TAC}} N : A.$$

(4.13) then follows by rules (Eq″) and ($\forall$ e).

Case (Eq″). Trivial by rule (Eq″).

Case (Eq′κ). Trivial by rule (Eq′κ).

Case ($\equiv_\alpha$). Trivial by rule ($\equiv'_\alpha$). ■

For the converse we have:

Theorem 4.19 *If Γ is a well-formed environment, and if* (4.13) *holds, then* (4.12) *holds.*

Proof By induction on the proof of (4.13).

Basis: If (4.13) is axiom (P T), then (4.12) follows by axiom (P T).

Induction step: The cases are by the last rule in the deduction of (4.13).

Case ($\kappa\kappa'$Formation). (4.13) is

$$\Gamma \vdash_{\mathrm{TAC}} ((\forall x : A))B : \kappa',$$

where x does not occur free in A or in Γ. The premises are

$$\Gamma \vdash_{\mathrm{TAC}} A : \kappa \quad \text{and} \quad \Gamma, x : A \vdash_{\mathrm{TAC}} \Gamma : \kappa'.$$

Hence, $\Gamma, x : A$ is a well-formed environment (with respect to TAC), and so by the induction hypothesis

$$\Gamma, x : A \vdash_{\mathrm{TACS}} \Gamma : \kappa'.$$

Hence, (4.12) follows by (Pi).

Case ($\forall$ e). (4.13) is

$$\Gamma \vdash_{\mathrm{TAC}} MN : [N/x]B,$$

where the premises are

$$\Gamma \vdash_{\mathrm{TAC}} M : (\forall x : A)B \quad \text{and} \quad \Gamma \vdash_{\mathrm{TAC}} N : A.$$

By the induction hypothesis,

$$\Gamma \vdash_{\mathrm{TACS}} M : (\forall x : A)B \quad \text{and} \quad \Gamma \vdash_{\mathrm{TACS}} N : A.$$

Hence, (4.12) follows by rule ($\forall$ e).

Case ($\forall\kappa$i). (4.13) is

$$\Gamma \vdash_{\mathrm{TAC}} \lambda x{:}A \,.\, M : (\forall x : A)B,$$

where the premises are

$$\Gamma, x : A \vdash_{\mathrm{TAC}} M : \Gamma \quad \text{and} \quad \Gamma \vdash_{\mathrm{TAC}} A : \kappa,$$

where x does not occur free in A or in Γ. It follows that Γ, $x : A$ is a well-formed environment with respect to TAC, and so by the induction hypothesis,

$$\Gamma, x : A \vdash_{\mathrm{TACS}} M : B.$$

Hence, (4.12) follows by rule (λi).

Cases (Eq″), (Eq′κ), *and* ($\equiv'_\alpha$). Trivial by the corresponding rules in TACS. ■

Theorem 4.20 *A necessary and sufficient condition that* (4.12) *hold is that* Γ *be a well-formed environment* (*with respect to TAC*) *and that* (4.13) *hold.*[15]

Proof Theorems 4.18 and 4.19. ■

Corollary 4.20.1 *An environment Γ is well-formed with respect to TAC if and only if it is well-formed with respect to TACS.*

For this reason, we shall no longer specify the system with respect to which an environment is well-formed.

Remark The system TACS is slightly more general than the sequent version of the theory of constructions presented by Coquand and Huet in that its equality rules are more general. To obtain a natural deduction system equivalent to Huet's system, the rules (Eq$'\kappa$) must be deleted, rule (Eq$''$) must be replaced by the two more restricted rules

$$\frac{M : A \quad B : \kappa \quad A =_* B}{M : B,}$$

and rule ($\equiv'_\alpha$) must be generalized to allow changes of bound variables in both parts of a formula $M : A$. The corresponding changes in TACS include introducing equality rules corresponding to those given above, and modifying rule ($\equiv_\alpha$) accordingly.[16]

[15] Pottinger [Pot87] proposes a sequent formulation that is closer to TAC than is TACS and helps to emphasize the equivalence. In Pottinger's system, which he calls TOC 1, rules (Pi) and (Vi) are replaced, respectively, by Hyp ($\Gamma \vdash A : \kappa \Rightarrow \Gamma, x : A \vdash x : A$) and Reit ($\Gamma \vdash E \& \Gamma, F \vdash G \Rightarrow \Gamma, F \vdash E$). Pottinger proves that TOC 1 is equivalent to TACS (which he calls TOC 2). Since Pottinger's TOC 1 is a sequent version of TAC in the style of Fitch [Fit52], Pottinger's equivalence result can be considered another form of this theorem.

[16] Pottinger's TOC 1 (see the previous footnote) actually uses this more restricted version of the equality rules.

Chapter 5

REPRESENTING LOGIC AND MATHEMATICS IN THE THEORY OF CONSTRUCTIONS

It is now time to show that the theory of constructions can be a useful basis for the ROMULUS system, and to show that we can represent many important concepts from logic and mathematics in the theory.

This representation has actually been done by Coquand and Huet[1]. However, their presentation consists of little more than definitions and examples, and so a number of people have doubted the power of the theory. Here, in addition to the important definitions and examples, we shall look at some proof-theoretic consequences of the strong normalization theorem to show that these concepts behave the way we want them to.

We begin in Section 5.1 with the representation of propositional and predicate logic with equality. In Section 5.2 we discuss the addition of axioms to the system and how this might affect consistency. Then, in the remaining sections, we take up the representation of arithmetic, elementary set theory, and functions. The representation of arithmetic includes the axiom of mathematical induction, and it can thus serve as a model for the representation of inductively generated free algebras. As an example of this, we take up lists (finite sequences). These lists are useful in the formulation of the of the hook-up security property.

[1]See [CH86], [CH], and [Hue86], chapters 11 and 12.

5.1 Representing logic with equality

We have already discussed representing the connectives and quantifiers of logic in TAP (Section 2.4) and TAT (Section 3.6). Since TAP can be interpreted in the theory of constructions (by Theorem 4.2), we can use these same definitions. It will be convenient to repeat the appropriate definitions here. They are taken practically word-for-word from Section 3.6, but a notation more suggestive of logic will be used.

To use these definitions, we need the arrow, or function-space, type. This now becomes the implication proposition operator:

Definition 5.1 (Implication proposition operator) The term F is defined as follows:

$$\mathsf{F} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, (\forall x : u)v.$$

We use either $A \to B$ or $A \supset B$ as an abbreviation for $\mathsf{F}AB$, depending on the context.

It is easy to show that $\to$ satisfies the rules ($\to$ e)and ($\to$ i). This means, of course, that $\supset$ satisfies rules ($\supset$ e)and ($\supset$ i).

Definition 5.2 (Cartesian product proposition) The *conjunction proposition operator* and its associated pairing and projection operators are defined as follows:

(a) $\Lambda \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, (\forall w : \mathsf{Prop})((u \to v \to w) \to w)$;

(b) $\mathsf{D} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}u \,.\, \lambda y{:}v \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda z{:}u \to v \to w \,.\, zxy$;

(c) $\mathsf{fst} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}\Lambda uv \,.\, xu(\lambda y{:}u \,.\, \lambda z{:}v \,.\, y)$; and

(d) $\mathsf{snd} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}\Lambda uv \,.\, xv(\lambda y{:}u \,.\, \lambda z{:}v \,.\, z)$.

We use $A \wedge B$ as an abbreviation for ΛAB.

It is not at all difficult to prove from these definitions that if $A : \mathsf{Prop}$ and $B : \mathsf{Prop}$

$$\mathsf{D}AB : A \to B \to A \wedge B,$$

$$\mathsf{fst}AB : A \wedge B \to A,$$

and

$$\mathsf{snd}AB : A \wedge B \to B.$$

Furthermore, it is easy to see that if $M : A$ and $N : B$, then

$$\mathsf{fst}AB(\mathsf{D}ABMN) =_* M$$

and

$$\mathsf{snd}AB(\mathsf{D}ABMN) =_* N.$$

Definition 5.3 (Disjunction proposition operator) The *disjunction proposition operator* and its associated injection and case operators are defined as follows:
(a) $\mathsf{V} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, (\forall w : \mathsf{Prop})((u \to w) \to ((v \to w) \to w))$;
(b) $\mathsf{inl} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda x{:}u \,.\, \lambda w : \mathsf{Prop} \,.\, \lambda f{:}u \to w \,.\, \lambda g{:}v \to w \,.\, fx$;
(c) $\mathsf{inr} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda y{:}v \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda f{:}u \to w \,.\, \lambda g{:}v \to w \,.\, gy$; and
(d) $\mathsf{case} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}\mathsf{Prop} \,.\, \lambda z{:}\mathsf{V}uv \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda f{:}u \to w \,.\, \lambda g{:}v \to w \,.\, zwfg$.
We use $A \vee B$ as an abbreviation for $\mathsf{V}AB$.

It is easy to show that if $A : \mathsf{Prop}$ and $B : \mathsf{Prop}$, then

$$\mathsf{inl}AB : A \to A \vee B,$$

$$\mathsf{inr}AB : B \to A \vee B,$$

and

$$\mathsf{case}AB : A \vee B \to (\forall w : \mathsf{Prop})((A \to w) \to ((B \to w) \to w)).$$

Furthermore, it is easy to show that if $C : \mathsf{Prop}$, $M : A$, $N : B$, $F : A \to C$, and $G : B \to C$, then

$$\mathsf{case}AB(\mathsf{inl}ABM)CFG =_* FM$$

and

$$\mathsf{case}AB(\mathsf{inr}ABN)CFG =_* GN.$$

Definition 5.4 (False proposition) $\bot \equiv (\forall x : \mathsf{Prop})x$.

With regard to the existential quantifier, we are now in a position to remove an anomaly from Definition 3.16. For we now have the machinery to refer to functions whose values are types.

Definition 5.5 (Existential quantifier) The *existential quantifier proposition operator* and its associated pairing and projection functions are defined as follows:
(a) $\Sigma \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}u \to \mathsf{Prop} \,.\, (\forall w : \mathsf{Prop})((\forall x : u)(vx \to w) \to w)$;
(b) $\mathsf{D}' \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}u \to \mathsf{Prop} \,.\, \lambda x{:}u \,.\, \lambda y{:}vx \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda z{:}(\forall x : u)(vx \to w) \,.\, zxy$; and
(c) $\mathsf{proj} \equiv \lambda u{:}\mathsf{Prop} \,.\, \lambda v{:}u \to \mathsf{Prop} \,.\, \lambda w{:}\mathsf{Prop} \,.\, \lambda z{:}(\forall x : u)(vx \to w) \,.\, \lambda y{:}(\forall x : u)vx \,.\, ywz$.
We use $(\exists x : A)B$ as an abbreviation for $\Sigma A(\lambda x{:}A \,.\, B)$.

It not hard to show that if A : Prop and $B : A \to$ Prop, then

$$(\exists x : A)B : \text{Prop},$$

$$\mathsf{D}'AB : (\forall u : A)(Bu \supset (\exists x : A)(Bx)),$$

and

$$\text{proj}AB : (\forall x : A)((\forall u : A)(\forall v : Bu)x \supset (\exists w : A)(Bw) \supset x).$$

Furthermore, if in addition C : Prop, $M : A$, $N : BM$, and $Z : (\forall u : A)(Bu \to C)$, then

$$\text{proj}ABCZ(\mathsf{D}'ABMN) =_* ZMN.$$

Note that D′ differs from D only in the types postulated for some of the bound variables. But this difference is enough to make it impossible to define a right projection for D′ that is correctly typed[2].

We can also define equality over any type:

Definition 5.6 (Equality proposition) The *equality proposition*

$$M =_A N,$$

where A is assigned type Prop, is defined to be

$$\mathsf{Q}AMN,$$

where

$$\mathsf{Q} \equiv \lambda u{:}\text{Prop} \,.\, \lambda x{:}u \,.\, \lambda y{:}u \,.\, (\forall z : u \to \text{Prop})(zx \supset zy).$$

It is not hard to show that if A : Prop and $X : A$, then

$$\lambda x{:}A \to \text{Prop} \,.\, \lambda u{:}zX \,.\, u : X =_A X,$$

and that if in addition $Y : A$, $M : X =_A Y$, $Z : A \to$ Prop, and $N : ZX$, then

$$MZN : ZY.$$

This gives us the reflexive law of the equality proposition and the substitution property; these two properties are well known to imply all the usual properties of equality.

It is not hard to see from this that we have all the usual properties of constructive predicate logic with equality.

[2]On this point, see [Car86]. Of course, fst works as a left projection function for D′.

We can also interpret classical logic. One interpretation[3] is based on the following easily proved facts about intuitionistic logic:

$$\vdash \neg\neg\neg A \supset \neg A,$$

$$\neg\neg A \supset A, \neg\neg B \supset B \vdash \neg\neg(A \wedge B) \supset (A \wedge B),$$

and

$$\neg\neg A(x) \supset A(x) \vdash \neg\neg(\forall x)A(x) \supset (\forall x)A(x).$$

Results corresponding to these can easily be proved in the theory of constructions. This means that for formulas A which are *classical*, that is for which $\vdash \neg\neg A \supset A$, the logic is classical. Furthermore, all negative formulas are classical and both $\wedge$ and $\forall$ preserve classical formulas. For other classical connectives and the existential quantifier, we can use their familiar classical properties to define them:

$$A \supset_c B \equiv \neg(A \wedge \neg B),$$

$$A \vee_c B \equiv \neg(\neg A \wedge \neg B),$$

and

$$(\exists_c x : A)B \equiv \neg(\forall x : A)\neg B.$$

Since these are all negative formulas, they are all classical.

It is not hard to prove that if A is classical (in a well-formed environment Γ), then there is a term M all of whose free variables are assigned types in Γ such that

$$\Gamma \vdash_{\mathrm{TAC}} M : \neg A \vee_c A.$$

If this method of representing classical logic is used in any "applied" theory, then it is necessary to be certain that

$$\neg\neg E \supset E$$

is provable for each formula E corresponding to an atomic formula in ordinary first order logic. To assure this, it may well be necessary to take these formulas as new axioms.

A second method of interpreting classical logic is as follows: define

$$\mathsf{Bool} \equiv (\forall u : \mathsf{Prop})(u \to u \to u),$$

[3]See [CH] §3.3, where this is done for propositional logic.

$$\mathsf{T} \equiv \lambda u : \mathsf{Prop} \,.\, \lambda x : u \,.\, \lambda y : u \,.\, x,$$

and

$$\mathsf{F} \equiv \lambda u : \mathsf{Prop} \,.\, \lambda x : u \,.\, \lambda y : u \,.\, y.$$

Here, Bool represents the boolean type familiar from the usual programming languages, and T and F for the familiar truth values. The familiar `if ... then ... else` operator is defined as follows:

$$\mathsf{Cond} \equiv \lambda u : \mathsf{Prop} \,.\, \lambda v : \mathsf{Bool} \,.\, \lambda x : u \,.\, \lambda y : u \,.\, vuxy.$$

It is easy to prove that T:Bool and F:Bool and, if A is any type in Prop and $M : A$ and $N : A$, then

$$\mathsf{Cond}A\mathsf{T}MN =_* M$$

and

$$\mathsf{Cond}A\mathsf{F}MN =_* N.$$

The propositional connectives familiar to most programmers can now be defined:

$$\neg_k \equiv \lambda x : \mathsf{Bool} \,.\, \mathsf{Cond}\ \mathsf{Bool}\ x\ \mathsf{F}\ \mathsf{T},$$

$$\wedge_k \equiv \lambda x : \mathsf{Bool} \,.\, \neg_k x\ \mathsf{Bool}\ \mathsf{F},$$

and

$$\vee_k \equiv \lambda x : \mathsf{Bool} \,.\, x\ \mathsf{Bool}\ \mathsf{T}.$$

It is then easy to prove the following:

$$\neg_k\mathsf{T} =_* \mathsf{F} \qquad \neg_k\mathsf{F} =_* \mathsf{T}$$

$$\wedge_k\mathsf{TT} =_* \mathsf{T} \qquad \wedge_k\mathsf{TF} =_* \mathsf{F}$$

$$\wedge_k\mathsf{FT} =_* \mathsf{F} \qquad \wedge_k\mathsf{FF} =_* \mathsf{F}$$

$$\vee_k\mathsf{TT} =_* \mathsf{T} \qquad \vee_k\mathsf{TF} =_* \mathsf{T}$$

$$\vee_k\mathsf{FT} =_* \mathsf{T} \qquad \vee_k\mathsf{FF} =_* \mathsf{F}$$

We can then get implication as usual by defining

$$\supset_k \equiv \lambda x : \mathsf{Bool} \,.\, \lambda y : \mathsf{Bool} \,.\, \neg_k(x \wedge_k \neg_k y),$$

and its usual truth table properties will follow.

In this formulation of classical logic, a proof of a proposition A is not a term with that proposition as its type, but rather a term with the type $A =_{\mathsf{Bool}} \mathsf{T}$. Thus,

unlike the first interpretation of constructive logic, this interpretation is based on a different set of terms to represent the propositions. In fact, it is based on the idea[4] that there are only two propositions, T and F.

Extending this second interpretation to quantifier logic is a bit complicated. The obvious way to proceed is to assume that we have a propositional function A over some domain D, which is a type. In this case, this means that $A : D \to \mathsf{Bool}$. We would want $(\forall_k x : D)(Ax)$ to be T if and only if AM is T for every $M : D$ and to be F otherwise; but this specification assumes classical logic, whereas the type

$$(\forall x : D)(Ax =_{\mathsf{Bool}} \mathsf{T})$$

is treated constructively by TAC, and in general there is no term with the type

$$(\forall x : D)(Ax =_{\mathsf{Bool}} \mathsf{T}) \vee (\exists x : D)(Ax =_{\mathsf{Bool}} \mathsf{F}).$$

One possible solution is to use the first interpretation of classical logic, and replace $\exists$ by $\exists_c$. But this will only work if D is a type for which there is a term of type

$$(\forall x : D)(\neg\neg Ax =_{\mathsf{Bool}} \mathsf{T} \supset Ax =_{\mathsf{Bool}} \mathsf{T}).$$

A third possible method of interpreting classical logic is to add a new axiom by assigning to an atomic constant the type

$$(\forall u : \mathsf{Prop})(\neg u \vee u).^5$$

We will have more to say about this in Section 5.2.

[4] Originally due to Frege.

[5] We could equally well use the formula $(\forall u : \mathsf{Prop})(\neg\neg u \supset u)$.

5.2 Adding axioms to the theory of constructions

As we have seen, when logic is represented in the theory of constructions, the formulas are all represented by types in Prop; the terms in these types will represent proofs. One consequence of this is that assuming a new axiom A will mean taking a new atomic constant c and adding $c : A$ as a new assumption to the environment.

Now the way we have proved the strong normalization theorem in Chapter 4 guarantees that such constants can be added without interfering with the proof of the theorem provided that these new constants do not occur at the heads of new redexes. But this is just the way new axioms are added. Thus, adding new axioms does not have any effect on the strong normalization theorem.

But adding new axioms may well affect the consistency of the system. Suppose, for example, we assume $c : \bot$. This amounts to assuming as an axiom $\bot$, i.e., to assuming the inconsistency of the system. This is one way in which the theory of constructions differs from the second order polymorphic typed λ-calculus: in the latter, Theorem 2.4 shows that the strong normalization theorem implies both the consistency of the entire system and of any set of assumptions[6], whereas in the former, as we have seen, the strong normalization theorem does not imply the consistency of all sets of assumptions.

The strong normalization theorem does, however, imply the consistency of the *empty* environment, and thus of the system TAC itself:

Theorem 5.1 (Consistency of TAC) *There is no closed term M such that*

$$\vdash_{\mathrm{TAC}} M : \bot.$$

Proof Similar to the proof of Theorem 2.4. ■

Note that this proves the consistency of the higher-order constructive and classical logic of the previous section.

Although the strong normalization theorem does not imply the consistency of all sets of assumptions, it does imply the consistency of some particular sets of assumptions. For example, suppose Γ is

$$x_1 : \neg A_1, x_2 : \neg A_2, \ldots, x_n : \neg A_n,$$

where $\neg A$ is defined to be $A \supset \bot$. To show that Γ is consistent it is sufficient to show that there is no closed term M for which

$$\Gamma \vdash_{\mathrm{TAC}} M : A_i$$

[6] Of course, if we allowed new constants in TAP, we would get the same sort of possibilities for inconsistency that we have in the theory of constructions.

for any i. As an example, let us prove that negations of equations between terms with distinct normal forms are consistent if there are no other assumptions.

Theorem 5.2 (Q-consistency[7]) *Let Γ be a set of assumptions in which each formula assigns to a rm (distinct) constant a type which converts to the form $\neg P =_A Q$ for terms P and Q of type A with distinct normal forms. Suppose that there is a closed term R such that*

$$\Gamma \vdash_{\mathrm{TAC}} R : M =_A N.$$

Then

$$M =_* N.$$

Proof Let $\mathcal{D}$ be a deduction in normal form of

$$\Gamma \vdash_{\mathrm{TAC}} R : M =_A N.$$

We proceed by induction on the structure of $\mathcal{D}$. Thus, we may suppose as part of the induction hypothesis that the theorem holds for any proper subdeduction of $\mathcal{D}$. Suppose that the last inference in $\mathcal{D}$ (except for equality rules) is by ($\forall$ e). Because $\mathcal{D}$ is normal, the only inferences in the left branch of $\mathcal{D}$ are ($\forall$ e) and (Eq''). Consider the formula at the top of the left branch of $\mathcal{D}$. Because of the form of $\mathcal{D}$ and of the rules of TAC, this formula is not a discharged assumption. If it is an undischarged assumption, then the term of that formula to which the type is assigned is a variable x, and $R =_* xR_1R_2 \ldots R_n$, contradicting the assumption that R is closed. If it is a formula of Γ, then the deduction of the minor (right) premise for the inference by ($\forall$ e) of which the formula in question is the major (left) premise is a proper subdeduction of $\mathcal{D}$ whose conclusion has the form $S : P =_A Q$ for a closed term S and terms P and Q with distinct normal forms, contradicting the assumption that the theorem holds for any proper subdeduction of $\mathcal{D}$. Hence, the last non-equality inference in $\mathcal{D}$ is not by ($\forall$ e).

Since

$$M =_A N \quad =_* \quad (\forall z : A \to \mathsf{Prop})(zM \supset zN),$$

it follows that that last non-equality inference is by ($\forall$Ti), $R \equiv \lambda z : A \to \mathsf{Prop} \,.\, P$, and $\mathcal{D}$ has the form[8]

[7] This term is due to Curry; see [CF58] §8E3, p. 270.

[8] Possibly modulo some manipulations involving rules (Eq'P), (Eq'T), and (Eq''); we will not bother to mention this fact again in what follows.

$$\dfrac{\begin{array}{c}1\\ [z : A \to \mathsf{Prop}]\\ \mathcal{D}_1(z)\\ P : zM \supset zN\end{array} \qquad \dfrac{\mathsf{Prop} : \mathsf{Type} \qquad A : \mathsf{Prop}}{A \to \mathsf{Prop} : \mathsf{Type}}\ (\forall\,\mathrm{e})}{\lambda z : A \to \mathsf{Prop}\ .\ P : (\forall z : A \to \mathsf{Prop})(zM \supset zN),}\ (\forall\mathrm{Ti}-1)$$

where z is a variable which does not occur free in Γ, M, or N. An argument similar to the above argument for $\mathcal{D}$ shows that the last non-eq inference in $\mathcal{D}_1(z)$ is not by ($\forall$ e), provided that at the end of the argument we note that although z may occur free in P, since z does not occur free in Γ it can only occur free in the discharged assumption, and the type assigned to z by that assumption makes it impossible for it to occur at the top of the left branch in $\mathcal{D}_1(z)$. Hence, the last non-eq inference in $\mathcal{D}_1(z)$ is by rule ($\forall$Pi), $P =_* \lambda w : zM\ .\ Q$, and $\mathcal{D}_1(z)$ has the form

$$\dfrac{\begin{array}{c}2\\ [w : zM]\\ \mathcal{D}_2(w)\\ Q : zN\end{array} \qquad \dfrac{z : A \to \mathsf{Prop} \qquad M : A}{zM : \mathsf{Prop}}\ (\to\,\mathrm{e})}{\lambda w : zM\ .\ Q : zM \supset zN,}\ (\forall\mathrm{Pi}-2)$$

where w is a variable distinct from z which does not occur free in Γ, M, or N. By an argument similar to that above, the last inference in $\mathcal{D}_2(w)$ is not by rule ($\forall$ e). Furthermore, any deduction of $Q : zN$ must use the hypothesis $w : zM$. Since $\mathcal{D}_2(w)$ is normal and zM and zN are simple types, it is not hard to see that the only rule that can occur in $\mathcal{D}_2(w)$ is (Eq″), from which it follows that $Q \equiv w$ and, more important, $M =_* N$. ■

Corollary 5.2.1 *If Γ is as in the theorem, then it is consistent; i.e., there is no closed term S such that*

$$\Gamma \vdash_{\mathrm{TAC}} S : \bot.$$

This theorem can be generalized somewhat. For example, if the types of the variables are suitably restricted to prevent substitution instances of P and Q which

are convertible to each other, it is presumably possible to prove a version of the theorem for universally quantified inequalities or for implications whose consequents are inequalities. Furthermore, as we shall see in the next section, it is possible to prove a similar theorem for a universally quantified inequality together with a universally quantified implication between equalities in which it can be shown that if the terms in the antecedent have distinct normal forms, then so do the terms in the consequent.

At the end of Section 5.1, we noted that we can obtain classical logic by taking $(\forall u : \mathsf{Prop})(\neg u \vee u)$ as a new axiom; i.e., by assuming

$$c : (\forall u : \mathsf{Prop})(\neg u \vee u),^{9}$$

for an atomic constant c. We need some evidence that adding this assumption does not introduce inconsistency. Of course, if we start with assumptions which are inconsistent with the law of the excluded middle, then adding this assumption will lead to a contradiction. But in most known systems without such assumptions, the consistency of the constructive version of the system is well-known to imply the consistency of the classical version. This makes it likely that adding this assumption to most consistent well-formed environments[10] will not make the environment inconsistent.

Remark We have looked here at adding constants that do not head redexes. In general, when we want a new redex, we define a closed term that can be shown by an ordinary β-reduction to head the required redex. This does not mean that using such a definition is the most efficient way to implement the system. It does, however, show that adding the new constant and reduction rule will not upset the strong normalization theorem, since any infinite reduction using the new constant and reduction rule will imply the existence of an infinite reduction from ordinary β-reduction using the closed term which can be shown to have the same reduction rule.

[9] Or, equally well, $c : (\forall u : \mathsf{Prop})(\neg\neg u \supset u)$.
[10] Which do not assign a type to c.

5.3 Representing arithmetic

As we saw in Section 2.4, we can easily represent the natural numbers in TAP. If this definition is modified for TAC, it becomes the following:

Definition 5.7 (Natural number type)
(a) $\mathsf{N} \equiv (\forall A : \mathsf{Prop})((A \to A) \to (A \to A))$;
(b) $\mathbf{0} \equiv \lambda A{:}\mathsf{Prop} \,.\, \lambda x{:}A \to A \,.\, \lambda y{:}A \,.\, y$;
(c) $\sigma \equiv \lambda u{:}\mathsf{N} \,.\, \lambda A : \mathsf{Prop} \,.\, \lambda x{:}A \to A \,.\, \lambda y{:}A \,.\, x(uAxy)$;
(d) $\pi \equiv \lambda u{:}\mathsf{N} \,.\, \mathsf{snd}_{\mathsf{N},\mathsf{N}}(u(\mathsf{N} \times \mathsf{N})\, Q(\mathsf{D}_{\mathsf{N},\mathsf{N}}\mathbf{00}))$,
where $Q \equiv \lambda v : \mathsf{N} \times \mathsf{N} \,.\, \mathsf{D}_{\mathsf{N},\mathsf{N}}(\sigma(\mathsf{fst}_{\mathsf{N},\mathsf{N}}v))(\mathsf{fst}_{\mathsf{N},\mathsf{N}}v)$; and
(e) $\mathsf{R} \equiv \lambda A : \mathsf{Prop} \,.\, \lambda x{:}A \,.\, \lambda y{:}\mathsf{N} \to A \to A \,.\, \lambda z{:}\mathsf{N} \,.\, z(\mathsf{N} \to A)P(\lambda w : \mathsf{N} \,.\, x)z$,
where $P \equiv \lambda v : \mathsf{N} \to A \,.\, \lambda w : \mathsf{N} \,.\, y(\pi w)(v(\pi w))$.
The term n, which represents the natural number n, is defined to be

$$\sigma(\sigma(...(\sigma\mathbf{0})...)),$$

where there are n occurrences of σ.

As we saw above, it is not hard to show that

$$\mathbf{0} : \mathsf{N},$$

$$\sigma : \mathsf{N} \to \mathsf{N},$$

$$\pi : \mathsf{N} \to \mathsf{N},$$

and

$$\mathsf{R} : (\forall A : \mathsf{Prop})(A \to (\mathsf{N} \to A \to A) \to \mathsf{N} \to A).$$

It is also easy to show that

$$\mathsf{n} =_* \lambda A : \mathsf{Prop} \,.\, \lambda x{:}A \to A \,.\, \lambda y{:}A \,.\, x(x(...(xy)...)),$$

where there are n occurrences of x after the last abstraction,

$$\pi\mathbf{0} =_* \mathbf{0},$$

$$\pi(\sigma\mathsf{n}) =_* \mathsf{n},$$

and also, for any type $A : \mathsf{Prop}$ and any terms M and N of types A and $\mathsf{N} \to A \to A$ respectively,

$$\mathsf{R}AMN\mathbf{0} =_* M,$$

and

$$\mathsf{R}AMN(\sigma \mathsf{n}) =_* N\mathsf{n}(\mathsf{R}AMN\mathsf{n}).$$

It is also not hard to show that

$$\mathsf{N} : \mathsf{Prop}.$$

We know that this definition works in the sense that we can define all primitive recursive functions and that the peano axioms hold. However, our knowledge of the peano axioms is entirely metatheoretic; we do not get the formulas representing these axioms as theorems of TAC. To get the peano axioms holding formally within TAC, we need to add some new axioms. The first two axioms we need are obvious:

$$\mathsf{Peano1} \equiv (\forall n : \mathsf{N})(\neg \sigma n =_{\mathsf{N}} \mathbf{0})$$

and

$$\mathsf{Peano2} \equiv (\forall m : \mathsf{N})(\forall n : \mathsf{N})(\sigma m =_{\mathsf{N}} \sigma n \supset m =_{\mathsf{N}} n).$$

We also need the induction axiom:

$$\mathsf{Peano} \equiv (\forall A : \mathsf{N} \to \mathsf{Prop})((\forall m : \mathsf{N})(Am \supset A(\sigma m)) \supset A\mathbf{0} \supset (\forall n : \mathsf{N})(An)).$$

Since the defining equations for + and × follow from the reduction properties of R and rule (Eq″), it may appear that we have everything we need for arithmetic.

However, we are not finished. For although the only closed terms of type N are known to be natural numbers[11], so that the axiom Peano does not really restrict the domain of objects in N, we do need to be able to talk about objects in other types which are not natural numbers. We may even want to create a supertype of N, and in such a supertype, where we will have things which are not natural numbers, we will want to be able to assert that an object is not a natural number. To do this, we need to be able to say that something *is* a natural number. And so far, we have no way of doing this that is part of the logic; we have only

$$M : \mathsf{N},$$

which is definitely *not* the same thing. Thus, we need a predicate of the logic, $\mathcal{N}$, which says that something is a natural number. The definition we want is as follows:

$$\mathcal{N} \equiv \lambda n : \mathsf{N} \,.\, (\forall A : \mathsf{N} \to \mathsf{Prop})((\forall m : \mathsf{N})(Am \supset A(\sigma m)) \supset A\mathbf{0} \supset An).$$

[11] Except for $\lambda A : \mathsf{Prop}\lambda x : A \to A \,.\, x$; this term is η-convertible to 1, but not β-convertible. But this term is not *really* something other than a natural number.

It is easy to prove

$$\begin{aligned} &\vdash_{\mathrm{TAC}} \quad \mathcal{N} : \mathsf{N} \to \mathsf{Prop}, \\ &\vdash_{\mathrm{TAC}} \quad M : \mathcal{N}\mathbf{0}, \\ &\vdash_{\mathrm{TAC}} \quad N : (\forall n : \mathsf{N})(\mathcal{N}n \supset \mathcal{N}(\sigma n)), \end{aligned}$$

for closed terms M and N.

Now that we have the definition of $\mathcal{N}$, we no longer need the axiom Peano, for it is easy to prove[12] that there is a closed term M such that

$$\vdash_{\mathrm{TAC}} \; M : (\forall A : \mathsf{N} \to \mathsf{Prop}) \\ ((\forall m : \mathsf{N})(Am \supset A(\sigma m)) \supset A\mathbf{0} \supset (\forall n : N)(\mathcal{N}n \supset An)).$$

While this is not exactly Peano, it is close enough for practical purposes[13].

This leaves us with the axioms Peano1 and Peano2. These two axioms appear to constitute a minor variation of the well-formed environment Γ of Theorem 5.2. In fact, a similar proof gives us the following result:

Theorem 5.3 (Q-consistency of arithmetic) *If* Γ *is*

$$c_1 : \mathsf{Peano1}, c_2 : \mathsf{Peano2},$$

and if

$$\Gamma \vdash_{\mathrm{TAC}} R : M =_A N,$$

where R *is a closed term,* A *is a type in* Prop, *and* M *and* N *are terms of type* A, *then*

$$M =_* N.$$

Corollary 5.3.1 *If* Γ *is as in the theorem, then it is consistent; i.e., there is no closed term* S *such that*

$$\Gamma \vdash_{\mathrm{TAC}} S : \bot.$$

The theory of arithmetic we have just seen is an excellent prototype for inductively generated free algebras, which can all be defined by similar methods[14]. It is not strictly necessary to have definitions for the types and constants involved: the

[12] This is not mentioned in [Hue86] or [Hue87].

[13] What Peano actually does is to say that the induction principle holds formally for the type N. We know metatheoretically that it holds for N, but without the axiom Peano, we do not have the result as a formal theorem of TAC. Since we do have that formal knowledge about $\mathcal{N}$, it is difficult to imagine circumstances in which this formal knowledge about N would be necessary.

[14] Cf. [BB84].

above theory would work just as well if N, 0, σ, and R are new atomic constants[15]. If we do take them as atomic constants, then Peano can be interpreted as saying that type N is assigned only to terms in the set $\mathcal{N}$, and so we are justified in concluding the consistency of the system with axiom Peano added.

As an example of an inductively generated free algebra, let us consider lists. In ROMULUS we will use lists to formulate the hook-up security property. To have lists of terms of type A, we need a type List which, when applied to A, forms the type List$A$ of lists of objects of type A. We also need the empty list, nil$A$, and the function cons$A$ of type $A \to \mathsf{List}A \to \mathsf{List}A$ which puts an object of type A at the front of a list of objects of type A to produce a new list of objects of type A. We will want to be able to define recursively functions on lists and objects of type A. For example, the function append which concatenates two lists, is defined as follows, where L_1 and L_2 are lists of type List$A$ and $M : A$:

$$\begin{aligned} \mathsf{append}A(\mathsf{nil}A)L_2 &\equiv L_2, \\ \mathsf{append}A(\mathsf{cons}AML_1)L_2 &\equiv \mathsf{cons}AM(\mathsf{append}AL_1L_2). \end{aligned}$$

To take another example, the function reverse which reverses the order of a list is defined by

$$\mathsf{reverse}AL \equiv \mathsf{flip}AL(\mathsf{nil}A),$$

where flip is defined by

$$\begin{aligned} \mathsf{flip}A(\mathsf{nil}A)L_2 &\equiv L_2, \\ \mathsf{flip}A(\mathsf{cons}AML_1)L_2 &\equiv \mathsf{flip}AL_1(\mathsf{cons}AML_2), \end{aligned}$$

To make definitions like this, we need a term which plays with respect to lists the role that R plays with respect to N.

It turns out to be possible to define List, nil, and cons so that these recursive definitions become possible:

$$\mathsf{List} \equiv \lambda A.\mathsf{Prop} \,.\, (\forall u : \mathsf{Prop})((A \to u \to u) \to u \to u),$$

$$\mathsf{nil} \equiv \lambda A : \mathsf{Prop} \,.\, \lambda B : \mathsf{Prop} \,.\, \lambda f : A \to B \to B \,.\, \lambda y : B \,.\, y,$$

$$\mathsf{cons} \equiv \lambda A : \mathsf{Prop} \,.\, \lambda x : A \,.\, \lambda l : \mathsf{List}A \,.\, \lambda B : \mathsf{Prop} \,.\, \lambda f : A \to B \to B \,.\, \lambda y : B \,.\, fx(lBfy).$$

[15] Of course, the reduction rules for R have to be postulated in this case. We can have confidence that there is no problem with the strong normalization theorem if these new constants are assumed precisely because we can define all of them as closed terms from which the reduction rules for R can be deduced.

The intention is that if $L =_* (x_1, x_2, \ldots, x_n)$ is a list in $\mathsf{List}A$, $f : A \to B \to B$, and $y : B$, then

$$LBfyfx_1(fx_2(\ldots(fx_ny))\ldots).$$

To show that this definition works, note that if $h : A \to B \to B$ and $M : B$, and if g is defined by

$$g \equiv \lambda l : \mathsf{List}A \,.\, lBhM,$$

then g has the properties

$$\begin{aligned} g(\mathsf{nil}A) \quad & M, \\ g(\mathsf{cons}AxL) \quad & hx(gL), \end{aligned}$$

for all $x : A$ and $L : \mathsf{List}A$. This function g allows us to define append, reverse, and such other list functions as length, mapcar, null, car, and cdr.

Just as we defined $\mathcal{N}$ corresponding to N, so we can define $\mathcal{L}$ corresponding to List. The definition is as follows:

$$\begin{aligned} \mathcal{L} \equiv \lambda A : \mathsf{Prop} \,.\, \lambda x : \mathsf{List}\ A \,.\, (\forall y : \mathsf{List}A \to \mathsf{Prop}) \\ ((\forall u : A)\ (\forall l : \mathsf{List}A)(\mathcal{L}Al \supset \mathcal{L}A(\mathsf{cons}Aul)) \supset \mathcal{L}A(\mathsf{nil}A) \supset \mathcal{L}x). \end{aligned}$$

It is then easy to prove

$$\vdash_{\mathrm{TAC}}\ \mathcal{L} : (\forall A : \mathsf{Prop})(\mathsf{List}A \to \mathsf{Prop}),$$

$$\vdash_{\mathrm{TAC}}\ M : (\forall A : \mathsf{Prop})(\mathcal{L}A(\mathsf{nil}A)),$$

$$\vdash_{\mathrm{TAC}}\ N : (\forall A : \mathsf{Prop})(\forall u : A)(\forall l : \mathsf{List}A)(\mathcal{L}Al \supset \mathcal{L}A(\mathsf{cons}Aul)),$$

and

$$\begin{aligned} \vdash_{\mathrm{TAC}}\ P : (\forall A : \mathsf{Prop})(\forall B : \mathsf{List}A \to \mathsf{Prop}) \\ ((\forall u : A)(\forall l : \mathsf{List}A)(Bl \supset B(\mathsf{cons}Aul)) \supset B(\mathsf{nil}A) \supset (\forall l : \mathsf{List}A)(\mathcal{L}l \supset Bl)), \end{aligned}$$

for some closed terms M, N, and P. This gives us the desired induction property on lists. All we still need are axioms corresponding to Peano1 and Peano2:

$$\begin{aligned} (\forall A : \mathsf{Prop})(\forall x : A)(\forall y : A)\ (\forall l : \mathsf{List}A)(\forall m : \mathsf{List}A) \\ (\mathsf{cons}Axl =_{\mathsf{List}A} \mathsf{cons}Aym \supset x =_A y \wedge l =_{\mathsf{List}A} m), \end{aligned}$$

and

$$(\forall A : \mathsf{Prop})(\forall x : A)(\forall l : \mathsf{List}A)(\neg \mathsf{cons}Axl =_{\mathsf{List}A} \mathsf{nil}A).$$

A modification of the proof of Theorem 5.3 shows that these two axioms are consistent.

5.4 Representing sets and functions

We spoke in the last section of the *predicate* $\mathcal{N}$ of natural numbers. But most mathematicians prefer to think of the *set* of natural numbers. This point of view is easily accommodated in the theory of constructions, since it is easy to think of a predicate as a set[16].

Thus, suppose we have some type U : Prop or U : Type. Then we may think of U as the current *universe*. *Sets* over U are defined to be predicates of type $U \to$ Prop. More formally, we may define

$$\mathsf{Set}_U \equiv U \to \mathsf{Prop}.$$

In terms of this definition, $\mathcal{N} : \mathsf{Set}_\mathsf{N}$ and, if A : Prop, $\mathcal{L}A : \mathsf{Set}_{\mathsf{List}A}$. If $A : \mathsf{Set}_U$, then we define $x \in A$ to be Ax. The set $\{x : U|E\}$ is defined to be $\lambda x : U \,.\, E$. Inclusion of set A in set B can be defined by

$$A \subseteq B \equiv (\forall x : U)(x \in A \supset x \in B)$$

and the corresponding equality by

$$A = B \equiv A \subseteq B \wedge B \subseteq A.$$

A special intensional equality on U can be defined as follows:

$$x \doteq y \equiv (\forall A : \mathsf{Set}_U)(x \in A \supset y \in A).$$

Many of the usual sets and set operations can be easily defined. For example:

$$\emptyset \equiv \{x : U|\bot\},$$

$$A \cap B \equiv \{x : U|x \in A \wedge x \in B\},$$

$$A \cup B \equiv \{x : U|x \in A \vee x \in B\},$$

and

$$\sim A \equiv \{x : U|\neg x \in A\}.$$

When no confusion results, we can leave out U and write $\{x|E\}$, Set, etc.

It is important to remember the constructive nature of the logic. This means that the set operations given above are not exactly like those in ordinary mathematics. For example, we have

$$A \subseteq \sim\sim A,$$

[16]This material is based on the work of Huet [Hue86], Chapter 12 and [Hue87].

but not, in general, the converse.

One operation on sets that we do not have here is the power set operation. For the power set of A, i.e. the set of all subsets of A, is defined by

$$\mathcal{P}A \equiv \lambda B : \mathsf{Set} \,.\, B \subseteq A,$$

and the type of $\mathcal{P}A$ is not Set, which is $A \to \mathsf{Prop}$, but instead $\mathsf{Set} \to \mathsf{Prop}$. Terms of type $\mathsf{Set} \to \mathsf{Prop}$ will be called *classes*, and we will give the formal definition

$$\mathsf{Class}_U \equiv \mathsf{Set}_U \to \mathsf{Prop}.$$

Since U can be replaced by Set_U, all set operations are also class operations. We can define other class operations, for example

$$\bigcap C \equiv \{x | (\forall A : \mathsf{Set})(CA \supset x \in A)\},$$

and

$$\bigcup C \equiv \{x | (\exists A : \mathsf{Set})(CA \wedge x \in A)\}.$$

We can also define the singleton in terms of classes:

$$\{x\} \equiv \bigcap(\lambda A : \mathsf{Set} \,.\, x \in A).$$

With these definitions,

$$\mathcal{N} : \mathsf{Set}_{\mathsf{N}}.$$

We know metatheoretically that the closed terms which are elements of the set $\mathcal{N}$ are exactly the closed terms of type N. Thus, the set $\mathcal{N}$ represents the type N in a special way. There is no known uniform method of defining sets to represent types for arbitrary types that does not require extra axioms[17].

Most mathematicians think of functions as sets of ordered pairs, but this conception is not really appropriate here. For we already have functions built into the theory of constructions as primitive. A function is simply a term assigned to a type of the form $(\forall x : A)B$. Functions can, of course, be elements of sets, especially if the sets correspond to types the way $\mathcal{N}$ corresponds to N. Since a set corresponding to a type A is a term of type $A \to \mathsf{Prop}$, a set of functions from type A to type B is a term of type $(A \to B) \to \mathsf{Prop}$. To say that a function $f : U \to U$ is a function from *set* A to *set* B, we use the type

$$(\forall x : U)(x \in A \supset fx \in B).$$

[17]It is, of course, possible to add an axiom of the form $\mathcal{A}M$ for each closed term $M : A$, where A is a type and $\mathcal{A}$ is the set intended to represent it, but many of these axioms are likely to upset the proof of strong normalization.

It follows that the set of functions from set A to set B is

$$\lambda f : U \to U \,.\, (\forall x : U)(x \in A \supset fx \in B).$$

If $f : U \to U$, then for A : Set we can define

$$\mathsf{Preserve}\, f\, A \equiv (\forall x : U)(x \in A \supset fx \in A).$$

In terms of this operator, the induction axiom Peano can be written as

$$\mathsf{Peano} =_* (\forall A : \mathsf{N} \to \mathsf{Prop})((\mathsf{Preserve}\, \sigma A) \supset \mathbf{0} \in A \supset (\forall n : \mathsf{N})(n \in A)),$$

and the definition of $\mathcal{N}$ as

$$\mathcal{N} =_* \lambda n : \mathsf{N} \,.\, (\forall A : \mathsf{N} \to \mathsf{Prop})(\mathsf{Preserve}\, \sigma A \supset \mathbf{0} \in A \supset n \in A).$$

This may help to show how to standardize the definition of inductively defined free algebras.

This much set theory is sufficient for most practical mathematical purposes, but from the point of view of a set theorist it is incomplete. Its major weakness is that if A is a set, $\mathcal{P}A$ is not a set but a class; in the standard set theories it is also a set. To make this a set, we would need to have Set include not only the terms in $U \to \mathsf{Prop}$ but also in $(U \to \mathsf{Prop}) \to \mathsf{Prop}$, $((U \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop}$, etc. This can be represented in the theory of constructions as follows:[18] first define

$$\begin{aligned} \mathsf{Set}_1 &\equiv U \to \mathsf{Prop}, \\ \mathsf{Set}_{n+1} &\equiv \mathsf{Set}_n \to \mathsf{Prop}. \end{aligned}$$

Then we want to introduce a new type Set which will be assigned to terms in any of the types Set_n. This requires that each type Set_n be a *subtype* of Set.

There is a general method of making type A a subtype of type B: it is to take as an assumption

$$\lambda x : A \,.\, x \;:\; A \to B.$$

From this assumption and $M : A$, we get $(\lambda x : A \,.\, x)M : B$, and clearly $(\lambda x : A \,.\, x)M$ represents the same object as M; in fact, it reduces to M. Assumptions of this form have not been considered so far in the theory of constructions, and cannot occur in well-formed environments. However, they have been considered in connection with ordinary type assignment; see [CHS72], pp. 453 and 304, where they are called *proper inclusions*. Furthermore, conditions under which these assumptions are compatible with the normal form theorem are given in [Sel77] Remark 2 p. 23. It is possible to extend condition (i) of that Remark to TAC:

[18] This is not done in [Hue86] or [Hue87].

Theorem 5.4 (Consistency of proper inclusions) *Let Γ be a well-formed environment, and let Γ' be a sequence of assumptions each of which has the form*

$$\lambda x : A \,.\, x : A \to B,$$

where B is an atomic constant, the assumption $B : \kappa$ occurs in Γ, and $B \to C$ is not a type in Γ' for any type C. Then any deduction of

$$\Gamma, \Gamma' \vdash_{\mathrm{TAC}} M : A$$

is strongly normalizable and both M and A have normal forms.

Proof We begin by proving that the required deductions are SN. Begin by replacing in each assumption in Γ' the term $\lambda x : A \,.\, x$ by a variable which does not occur free in either Γ or Γ', using a distinct variable for each such assumption. The resulting deductions are all SN by Theorem 4.14. Hence, the deductions in which we are interested, which are all obtained by substituting terms for variables, are also all SN.

Now let us consider the terms in these deductions. These terms may contain redexes of the form

$$(\lambda x : A \,.\, x)M.$$

A contraction will replace this redex by M. What we need to know is that this will not produce a new redex. This could only happen if the original redex occurred in a subterm of the form

$$(\lambda x : A \,.\, x)M N_1 N_2 \ldots N_n,$$

and since the type of

$$(\lambda x : A \,.\, x)M$$

is B, which is by hypothesis a new constant and hence not convertible to the form $(\forall y : C)D$, this is impossible. ■

Now, in order to interpret a set theory in which the power set of a set is a set, we need only define Set_n as indicated above for each $n \geq 1$, define Set to be a new atomic constant, assume Set : Prop or Set : Type, and then assume

$$\mathsf{Set}_n : \mathsf{Set}$$

for each $n \geq 1$[19]. It follows from what we have just proved that this is consistent; for Set is essentially the union of all the Set_n, and in any given deduction, it will be possible to replace Set by the union of a finite number of the Set_n and thus avoid using any new assumptions.

[19]This involves an infinite number of assumptions, but they can all be described in a finite manner, and so it is not unreasonable to suppose that this can be implemented.

Appendix A

LIST OF POSTULATES AND SYSTEMS

Here are listed the various postulates which have appeared in this document and the systems in which they occur. A list of the systems and the number of their definitions is given in appendix 2. The rules are listed in the order in which their main operators first appear.

($\rightarrow$ Formation): TAJ, TAT

($\rightarrow$ e): TA, TAP, TAJ, TAT

($\rightarrow$ i): TA, TAP; (alternate form) TAJ, TAT

($\forall$ Formation): TAGU

($\forall$e): TAP; (another sense) NJ*; (another sense) TAGU, TAC; (another sense) TACS

($\forall$i): TAP; (another sense) NJ*; (another sense) TACS

($\forall$JFormation): TAJ

($\forall$Je): TAJ

($\forall$Ji): TAJ

($\forall$Pi): TAC

($\forall$Ti): TAC

($\forall$Ui): TAGU

($\forall\alpha$Formation): TAT

($\forall\alpha$e): TAT, TAG

($\forall\alpha$i): TAT, TAG

($\equiv_\alpha$): TA; (another sense) TACS
($\equiv'_\alpha$): TAP, TAJ, TAT, TAG, TAGU, TAC
($\equiv''_\alpha$): TAP
($\equiv''' \alpha$): TAJ, TAT
($\supset$ e): NA ($\subset$) NJ, NJ*
($\supset$ i): NA ($\subset$), NJ, NJ*
($\wedge$e): NJ, NJ*
($\wedge$i): NJ, NJ*
($\vee$e): NJ, NJ*
($\vee$i): NJ, NJ*
($\neg$e): Derived in NJ, NJ*
($\neg$i): Derived in NJ, NJ*
($\perp$ j): NJ, NJ*
($\perp$ j_θ): added to extended TA
($\perp$ $j\varphi_i$): TAJ
($\exists$e): NJ*
($\exists$i): NJ*
($\exists$JFormation): TAJ
($\exists$Je): TAJ
($\exists$Ji): TAJ
(e_i): TAJ
(ω_i): TAJ
(φ_i): TAJ
(void): TAJ
($\times$Formation): TAJ
($\times$e)$_1$: TAJ
($\times$e)$_2$: TAJ
($\times$i): TAJ
(+Formation): TAJ
(+e): TAJ
(+i)$_1$: TAJ
(+i)$_2$: TAJ
(Eq$''$): TAG, TAGU, TAC, TACS

(Eq'U): TAGU
(Eq'P): TAC, TACS
(Eq'T): TAC, TACS
(Pe): TACS
(Pi): TACS
(PPFormation): TAC
(PT): TAC; (another sense) TACS
(PT Formation): TAC
(TP Formation): TAC
(TT Formation): TAC
(λi): TACS

Appendix B

SYSTEMS AND THEIR DEFINITIONS

Here is a list of systems and their definitions.

NA($\supset$): Definition 3.2.

NJ: Definition 3.4.

NJ*: Definition 3.6.

TA: Definition 2.1.

Extended TA: Remark after Corollary 2.2.3 (end of Section 2.1).

TAC: Definition 4.2.

TACS: Definition 4.21.

TAG: Definition 2.22.

TAGU: Definition 2.24.

TAJ: Definition 3.10.

TAP: Definition 2.12.

TAT: Definition 3.12.

Bibliography

[BB84] Corrado Böhm and A Berarducci. Automatic synthesis of typed lambda-programs on term algebras. May 1984. Unpublished notes.

[Bee85] M. Beeson. *Foundations of Constructive Mathematics.* Springer, Berlin, 1985.

[C*86] R. Constable et al. *Implementing Mathematics with the Nupri Poof Development System.* Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[Car86] Luca Cardelli. *A Polymorphic λ-calculus with Type : Type.* Technical Report, Systems Research Center of Digital Equipment Corporation, Palo Alto, California, May 1986.

[CF58] Haskell Brooks Curry and Robert Feys. *Combinatory Logic.* Volume 1, North-Holland Publishing Company, Amsterdam, 1958. Reprinted 1968 and 1974.

[CH] Thierry Coquand and Gérard Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. Colloque de Logique, Orsay (July 1985), North Holland, forthcoming.

[CH84] Thierry Coquand and Gérard Huet. A theory of constructions. June 1984. Presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis.

[CH86] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL85*, pages 151–184, Springer-Verlag, Berlin, 1986.

[CHS72] Haskell Brooks Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic.* Volume 2, North-Holland Publishing Company, Amsterdam and London, 1972.

[Chu40] Alonzo Church. A formalization of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Coq] Thierry Coquand. Metamathematical investigations of a calculus of constructions. Received February 9, 1987.

[Coq85] Thierry Coquand. *Une Théorie des Constructions.* PhD thesis, University of Paris VII, 1985.

[Coq86a] Thierry Coquand. An analysis of Girard's paradox. In *Symposium on Logic in Computer Science*, pages 227–236, IEEE Computer Society, IEEE Computer Society Press, 1986.

[Coq86b] Thierry Coquand. A calculus of constructions. November 1986. Privately circulated.

[CR36] Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[Cur63] Haskell Brooks Curry. *Foundations of Mathematical Logic.* McGraw-Hill Book Company, Inc., New York, San Francisco, Toronto, and London, 1963. Reprinted by Dover, 1977 and 1984.

[Daa80] Diederik Ton van Daalen. *The Language Theory of AUTOMATH.* PhD thesis, Technische Hogeschool Eindhoven, February 1980.

[Fit52] Fredric Brenton Fitch. *Symbolic Logic.* The Ronald Press Company, New York, 1952.

[FLO83] S. Fortune, Daniel Leivant, and Michael J. O'Donnell. The expressiveness of simple and second order type structures. *Journal of the Association for Computing Machinery*, 30:151–185, 1983.

[Gen34] Gerhard Gentzen. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934. Translated in Sabo (ed.), *The Collected Papers of Gerhard Gentzen* as "Investigations into Logical Deduction".

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, North-Holland, Amsterdam, 1971.

[GMW79] M. J. Gordon, J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Springer Verlag, 1979. Lecture Notes in Computer Science 78.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. Roger Hindley and Jonathan P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–

490, Academic Press, New York, 1980. A version of this paper was privately circulated in 1969.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[Hue86] Gerard Huet. Formal structures for computation and deduction. May 1986. Course Notes, Carnegie-Mellon University, First Edition.

[Hue87] Gerard Huet. Inductionprinciples formalized in the calculus of constructions. In *Springer Lecture Notes in Computer Science 249*, pages 276–286, Springer-Verlag, 1987.

[Jas34] Stanisław Jaśkowski. On the rules of supposition in formal logic. *Studia Logica*, 1:5–32, 1934.

[Mar71a] Per Martin-Löf. *Hauptsatz* for the theory of species. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 217–233, North-Holland Publishing Company, Amsterdam and London, 1971.

[Mar71b] Per Martin-Löf. A theory of types. February 1971. Revised October 1971. Privately circulated.

[Mar73] Per Martin-Löf. *Hauptsatz* for intuitionistic simple type theory. In Patrick Suppes, Leon Henkin, Athanase Joja, and Gr.C. Moisil, editors, *Logic, Methodology, and Philosophy of Science IV*, pages 279–290, International Congress for Logic, Methodology, and Philosophy of Science, Bucharest, 1971, North-Holland Publishing Company, Amsterdam and London, 1973.

[Mar75] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, North-Holland Publishing Company, Amsterdam, 1975.

[Mar82] Per Martin-Löf. Constructive mathematics and computer science. In L. J. Cohen, J. Los, H. Pfeiffer, and K.-P. Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, pages 153–175, North-Holland Publishing Company, Amsterdam, 1982.

[Mar84] Per Martin-Löf. Intuitionistic type theory. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[Mil85] R. Milner. The standard ML core language. *Polymorphism*, 2, 1985.

[Mit86] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions (summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 308–319, 1986.

[Pot87] Garrel Pottinger. Two formulations of the theory of constructions. January 1987. Technical report in preparation, Odyssey Research Associates.

[Pra65] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Stockholm, Göteborg, and Uppsala, 1965.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Springer Lecture Notes in Computer Science 19*, pages 408–425, Springer-Verlag, 1974.

[Rey84] J. C. Reynolds. Polymorphism is not set-theoretic. In *Springer Lecture Notes in Computer Science 173*, pages 145–156, Springer-Verlag, 1984.

[Ros84] J. B. Rosser. Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6:337–339, 1984.

[Sel77] J. P. Seldin. A sequent calculus for type assignment. *Journal of Symbolic Logic*, 42:11–28, 1977.

[Ste72] Sören Stenlund. *Combinators, Lambda-Terms and Proof Theory.* D. Reidel, Dordrecht, Holland, 1972.

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.